Type System for Resource Bounds with Type-Preserving Compilation

by

Peng Wang

B.E., Tsinghua University (2010)
M.S., Tsinghua University (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2019

© Massachusetts Institute of Technology 2019. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
October 12, 2018

Certified by . . . .
Adam Chlipala
Associate Professor of Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A.Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Type System for Resource Bounds with Type-Preserving Compilation

by

Peng Wang

Submitted to the Department of Electrical Engineering and Computer Science
on October 12, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

This thesis studies the problem of statically bounding the resource usage of computer programs, from programs written in high-level languages to those in assembly languages. Resource usage is an aspect of programs not covered by conventional software-verification techniques, which focus mostly on functional correctness; but it is important because when resource usage exceeds the programmer's expectation by a large amount, user experience can be disrupted and large fees (such as cloud-service fees) can be charged. I designed TiML, a new typed functional programming language whose types contain resource bounds; when a TiML program passes the typechecking phase, upper bounds on its resource usage can be guaranteed. TiML uses indexed types to express sizes of data structures and upper bounds on running time of functions; and refinement kinds to constrain these indices, expressing data-structure invariants and pre/post-conditions. TiML's distinguishing characteristic is supporting highly automated time-bound verification applicable to data structures with nontrivial invariants. Type and index inference are supported to lower annotation burden, and, furthermore, big-O complexity can be inferred from recurrences generated during typechecking by a recurrence solver based on heuristic pattern matching.

I also designed a typed assembly language with resource bounds, and a type-preserving compiler that compiles well-typed TiML programs into well-typed assembly programs, conforming to the same bounds. Typechecking at the assembly level reestablishes the soundness of the bounds, and the types can serve as resource-usage certificates for the assembly programs.

I used Ethereum smart contracts as a real-world application of the techniques developed in this thesis. The assembly language I designed, TiEVM, is a typed version of the Ethereum Virtual Machine (EVM) bytecode language. I will demonstrate that TiML can be used as a new language to write smart contracts, and the generated TiEVM code is equipped with types proving that its resource usage – "gas" in Ethereum terminology – is bounded.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Computer Science

Acknowledgments

Looking back at my six PhD years, I feel lucky to have had Adam as my advisor at every junction: admitting me as his PhD student, holding my hand through my first PL publication, letting me pursue a research topic that I was passionate about but did not necessarily fit into his bigger agenda, and finally guiding me through a relatively smooth graduation process. His opinions sometimes sound nonsensical to me, but always thought-provoking, both on research and on life.

Thanks to Di Wang who co-authored the OOPSLA paper [78] with me. Part of this dissertation's content is based on the paper. As an undergraduate-student intern in our group, he pioneered the type-preserving compiler with resource bounds. I was helped in my design of the compiler by the reference [63] he directed me to and the idea of universally quantifying the cost of the continuation for functions in CPS.

I made many friends here at MIT who are both exceptional researchers and remarkable human beings: Zhengdong Zhang, Wenzhen Yuan, Ling Ren, Yu Zhang, Chiyuan Zhang, and Tianfan Xue. We supported each other in the darkest days of our PhD years, and all made it to the end.

To my parents (borrowing the words from Chris Okasaki's PhD dissertation): who would have thought on that first day of school that I would still be in school 24 years later?

Contents

1 Introduction **15**

1.1 Motivations for resource-usage analysis 15

1.2 Static approaches 16

1.2.1 Type-based approach 17

1.3 An application: the Ethereum platform 18

1.4 Elements of the thesis 21

1.4.1 TiML: the source language 21

1.4.2 TiEVM: the target language 22

1.4.3 The type-preserving compiler 25

1.5 Novelties 26

1.6 Notations 27

1.7 Source code 27

2 TiML **29**

2.1 TiML examples 29

2.2 Syntax and semantics 34

2.2.1 Syntax 34

2.2.2 Operational semantics 37

2.3 Type system 38

2.3.1 Typing rules 40

2.3.2 Typing examples 44

2.3.3 Soundness theorem 46

2.3.4 Decidability 46

2.4 Typechecker implementation and big-O inference 47
2.5 Formal soundness proof 49
2.6 ETiML: a TiML variant for smart contracts 54

3 TiEVM **59**
3.1 An EVM primer 59
3.2 Design of TiEVM 61
3.3 Syntax 62
3.4 Typing rules 66
3.4.1 Notations and conventions 66
3.4.2 Sequences and jumps 68
3.4.3 Basic blocks and whole programs 72
3.4.4 Stack manipulation and simple arithmetic 72
3.4.5 Memory access 74
3.4.6 Tuple and array initialization 77
3.4.7 Storage access 79
3.4.8 Miscellaneous 86

4 The type-preserving compiler **87**
4.1 Surface-TiML to μTiML 88
4.1.1 Surface-TiML 89
4.1.2 Translating into μTiML 92
4.2 CPS conversion 101
4.2.1 Type translation 101
4.2.2 Term translation 103
4.3 Closure conversion 107
4.4 Code generation 108
4.5 Derived cost models 119
4.5.1 TiEVM cost model 119
4.5.2 TiML cost model before code generation 119
4.5.3 TiML cost model before closure conversion 122

4.5.4 TiML cost model before CPS conversion 123
4.5.5 Surface-TiML cost model 125

5 Evaluation 127
5.1 Typechecking classic algorithms 127
5.2 Compiling smart contracts 133

6 Related Work 145
6.1 Dependent ML 145
6.2 AARA and RAML 146
6.3 Program logics and verification systems 147
6.4 Sized types and refinement types 148
6.5 Program analysis 149
6.6 Gas analysis for Ethereum 150
6.7 Other resource-analysis systems 150

A Technical details for TiEVM 153
A.1 TiEVM instructions (full list) 153
A.2 Expansions of TiEVM pseudo-instructions 154

B Technical details for the compiler 157
B.1 Cost definitions 157
B.2 CPS cost adjustments 159

List of Figures

1-1 Notations 27

2-1 TiML example: definition of list and fold-left 29
2-2 TiML example: merge sort 32
2-3 TiML example: red-black trees 33
2-4 TiML syntax 34
2-5 Operators 35
2-6 Definitions in operational semantics 38
2-7 Operational semantics 39
2-8 Typing contexts 39
2-9 Sorting rules 40
2-10 Typing rules 41
2-11 Typing rules (continued) 42
2-12 Configuration typing 45
2-13 ETiML example: token 55

3-1 TiEVM syntax 63
3-2 TiEVM typing contexts 66
3-3 TiEVM typing rules (sequences and jumps) 69
3-4 Typing rules for TiEVM programs and basic blocks 71
3-5 TiEVM typing rules (stack manipulations and simple arithmetic) . . 73
3-6 TiEVM typing rules (memory access) 74
3-7 TiEVM typing rules (tuple and array initialization) 77
3-8 TiEVM typing rules (storage access) 80

3-9 TiEVM typing rules (miscellaneous) 86

4-1 Syntax of Surface-TiML 89
4-2 Surface TiML kinding, typing, and declaration-checking rules (selected) 91
4-3 CPS conversion for types 101
4-4 CPS conversion for terms 104
4-5 Closure conversion 108
4-6 Code generation for types 109
4-7 Code generation for terms 110
4-8 Code generation for terms (continued) 111
4-9 Code generation for terms (continued) 112
4-10 Code generation for terms (continued) 113
4-11 Code generation for terms (outputing instruction sequence) 115
4-12 Code generation for terms (outputing instruction sequence, continued) 116
4-13 Code generation for terms (outputing instruction sequence, continued) 117
4-14 Code generation for top-level functions and programs 118

5-1 Typechecking time 129
5-2 Number of lines of code 130
5-3 Gas-estimation accuracy 139
5-4 TiML vs. Solidity 140
5-5 Same as Figure 5-4, except that the two functions with the largest slow-downs have been removed. 141
5-6 Typechecking and compilation time 144

List of Tables

5.1 Benchmarks 128
5.2 Benchmark contracts and their descriptions 134
5.3 The functions that have been measured and scenarios in which the functions are invoked 136
5.4 Evaluation results on the 8 benchmark smart contracts 138
5.5 Typechecking and compilation time 142

Chapter 1

Introduction

This chapter sets the stage for the thesis. It starts by discussing why we should care about a program's resource usage, more so than just doing casual big-O analysis or profiling the program on several test runs. It zooms in on one particular application, the Ethereum blockchain platform, as my main use case throughout the thesis. I will argue that knowing resource bounds statically (at compile time) has many benefits over runtime techniques, and I will describe available technical approaches to establish bounds. Among the available approaches, I will discuss why I choose a type-based approach, with its strengths and weaknesses. The last part of this chapter will highlight the novelties and briefly describe the structure of this thesis work.

1.1 Motivations for resource-usage analysis

Software verification has been mostly focused on functional correctness, which typically means the input and output (sometimes also the side effects) of a program are in relation according to some high-level specification. These specifications and proofs usually leave out some quantitative properties of the program such as how many CPU cycles and how much memory space it uses. Nonetheless, these nonfunctional aspects of programs are often of equal importance with functional properties in the everyday use of the program. Take the program's execution time for example. Mismatches between intended and actual execution time, manifested as "performance

bugs" [53, 67, 55], can frustrate users and/or cause serious security vulnerabilities [29]. As an example, [67] reports a performance bug in a mature Java application (JFreeChart) where a rendering function takes $O(n^2)$ time to draw a dataset while the inner iteration can be easily replaced by a memoization variable, cutting the time to $O(n)$. Unexpected freezing of a UI or waiting for a response greatly disrupts the user experience.

Sometimes resource usage directly determines the economic cost of the software. Two such examples are Amazon Cloud Services (AWS) and the Ethereum platform. Lambda, a Function-as-a-Service offering among AWS's services, charges users by the number of cloud-function invocations their client programs make, and the time and memory usage of those functions, hence for a client program that uses Lambda, the number and cost of such invocations are crucial economic metrics to consider. The Ethereum platform will be discussed in detail in Section 1.3.

1.2 Static approaches

The most commonly used dynamic techniques to control software quality are testing and profiling. Profiling is a manual debugging tool that is not suited for automatic quality control. Testing is ill-equipped to check time complexity for several reasons. (1) Wrong asymptotic complexities often manifest themselves only under large inputs, making test suites time-consuming and costly to run. The small input hypothesis [67, 61] which underlies most software-testing methodologies does not hold in performance testing. (2) Common testing methods such as assertions are hard to adopt for performance testing. (3) Unlike functional bugs, performance bugs are not fail-stop, making it hard even to tell whether a performance bug manifests itself. A human judgment call is often required to tell when performance goes awry. For all these reasons, static guarantees of time and more broadly resource usage become desirable.

In contrast to runtime techniques such as testing and profiling, static techniques aim to determine program properties at compile time. They achieve this by inspecting

the source code of the program, simulating the program behavior symbolically, and/or relying on programmer-provided annotations such as types. Because they do not focus on specific runs of the program with specifics set of inputs, they can usually derive program properties that will hold for all inputs in all circumstances. The guarantees one obtains from static techniques, therefore, are usually stronger than from testing and profiling. Additionally, the speed of a static analysis usually does not depend on the program's execution time (it will depend on other factors such as the size of the source code and the complexity of the analysis algorithm), so it avoids the time cost of testing programs on large inputs.

1.2.1 Type-based approach

There are a number of techniques for delivering static guarantees for programs, including symbolic evaluation, type systems, program logics, etc. I choose a type-system approach for its multiple advantages. (1) Type systems are good at handling higher-order programs, since functions are treated as first-class citizens just like other values.[1] (2) Types can serve as specifications, which make the code a self-contained unit with both specification and implementation. (3) Some type systems such as dependent type systems and refinement type systems allow complex predicates in types, which can be used as preconditions, postconditions, and invariants, crucial for analyzing complex data structures and algorithms. (4) Annotations give programmers an opportunity to help the analysis, and types are a principled way of annotating. (5) Types of API calls are important documentation to characterize the APIs' behavior. To quote [53], "Two thirds of the studied bugs are introduced by developers' wrong understanding of workload or API performance features. More than one quarter of the bugs arise from previously correct code due to workload or API changes. To avoid performance bugs, developers need performance-oriented annotation systems and change-impact analysis." (6) Type systems are enforceable disciplines that force programmers to think harder and more clearly about their code. From my personal

[1] Recognizing that a function is just a value of an ordinary type (the "arrow type") is the largest contribution the functional-programming paradigm made to the outer world, in my opinion.

experience, rewriting some existing Ethereum smart contracts using my new language with resource bounds in types forced me to think about why these contracts' resource costs are bounded, and it led me to uncover some lax thinking in the original code that causes unbounded resource usage, such as copying arbitrary-length strings between memory and storage.

Most existing work on static resource-usage analyses falls into one of two camps, the first of which aims at full automation, while the second aims at expressiveness (much like the split in the broader software-verification literature). When it comes to ease of use, nothing beats push-button systems, though at the cost of restricting domains to e.g. polynomial bounds [46] or first-order imperative programs [40]. Tools in this category also disallow user-provided hints when automation fails. The second camp aspires to verify hard programs against rich specifications [25], using techniques such as program logics and tools such as proof assistants, at the cost of writing proofs manually. A class of middle-way approaches recently gained popularity in software verification, pioneered by DML [80] and popularized by liquid types [70] and Dafny [57], whose central theme is to restrict the power of dependent types or program logics in exchange for some degree of automation. My work will be in the same spirit, where I ask the programmer to help by providing annotations, and I then try to make the typechecking and compiling experience as smooth as possible and, in case of failure, give useful feedback to help the programmer tweak annotations.

1.3 An application: the Ethereum platform

A large portion of this thesis work focuses on applying type-based techniques to writing smart contracts on the Ethereum platform. A large amount of technical work has been devoted to solve Ethereum-specific issues, so I will zoom in on the Ethereum platform in this section and describe it in detail.

Ethereum belongs to a set of technologies known as "blockchain technologies," which have attracted a large amount of public attention in recent years. A "blockchain" refers to a distributed ledger that book-keeps a history of transactions which are

agreed upon among a group of participants (fixed or fluid). These transactions can be seen as changes of the state of a certain virtual world. In Bitcoin [3], the original blockchain application, the world state is the balances of all accounts denoted in a virtual currency called "bitcoin," and a transaction is a sum-preserving change of balances (i.e. a transfer of bitcoins). A block is a set of transactions grouped together, and the history of transactions is represented as "a chain of blocks." Ethereum [4] extends Bitcoin by extending the world state to be a general mapping from an account number and an integer address to an integer value, and extending the transactions to be any changes of the state. It describes these transactions in a Turing-complete bytecode language called the Ethereum Virtual Machine (EVM) language.

The ability to express arbitrary computation opens Ethereum to many potential uses beyond cryptocurrency. For examples, it can be used to form a voting congress, to record shares of a corporation, to bookkeep the ownership of physical assets, and to store the supply-chain information of goods.

Because Ethereum transactions will be executed on each participating node, a transaction that loops forever or takes too much time or memory will slow down all the nodes, and the total waste of energy caused by unthoughtful EVM bytecode is considerable. To regulate time and memory consumption, EVM incorporates a "gas" mechanism. Each EVM instruction costs a certain amount of gas. A transaction's total amount of gas consumption will be the fee paid to the transaction processors ("miners"). Gas price in terms of Ethereum's virtual currency "ether" is determined in the market between transaction producers and processors.

Gas is paid upfront. The person that publishes a transaction needs to guess an upper bound of the transaction's total gas cost and pay for it at the beginning of the transaction. Unused gas will be refunded at the end of the transaction. If the transaction runs out of gas in the middle of execution, the whole transaction is discarded (state rolled back but gas not refunded).

The benefits of static techniques for controlling resource usage apply here. They are particularly advantageous over the alternative methods used in the Ethereum community today. At present, people come up with this upfront gas estimation in

two ways. One is to use the maximal figure possible (there is a gas limit imposed by the Ethereum protocol) and rely on the refunding of unused gas. There are two problems with this approach. One is that it requires the transaction issuer to have enough ethers to pay for the large gas amount upfront. Secondly, due to a design flaw in the current version of the Ethereum protocol ("Homestead"), any exception during the execution of the transaction will cause all gas to be confiscated. To make matters worse, smart-contract authors routinely use exceptions to discard transactions with illegal inputs, since such pass-or-throw checks make the code clean. An unmindful user who sends a wrong input to such a contract will lose all the gas paid upfront for this transaction.

Another gas-estimation approach is to dry-run the transaction off-chain to obtain the gas usage. This is a practical solution albeit with two drawbacks. One is that the result of the dry-run may be different from the actual on-chain run, if the execution depends on some on-chain factors such as the block-number/time-stamp of the current block or the hash of the previous five blocks. The second is that such an approach only gives gas usage of the smart contract on a particular input, not the gas-usage characteristic of the smart contract itself. Like a computer program's time complexity, such a characteristic should be described as a cost function on input size.

I aim to check and guarantee gas bounds statically and rely on type systems to do the checks. With type systems that are proven sound, we get formal guarantees that these bounds are respected under all circumstances. I designed two such type systems, one for a high-level functional language in which smart-contract authors write their programs, and one for EVM bytecode at the assembly level. A type-preserving compiler is developed to connect the two. This two-type-system approach moves the compiler out of the trusted computing base (TCB). One only needs to trust the soundness of the EVM type system to believe that a well-typed EVM program never uses more gas than specified in its types.

1.4 Elements of the thesis

This thesis work consists of three parts: a new high-level functional language called TiML, a new assembly-level language called TiEVM, and a type-preserving compiler translating well-typed TiML programs into well-typed TiEVM programs. Each of the three constituents will be introduced briefly in this section.

1.4.1 TiML: the source language

TiML (Timed ML) is an ML-like functional language with a type system that bounds time and memory. It uses indexed types to represent data sizes and resource bounds, and it uses dependent sorts to put constraints on these indices. It supports algebraic datatypes whose indexing schemes (i.e. size metrics) can be chosen freely by the programmer. Dependent sorts let one write complex pre-/post-conditions and invariants of algorithms and data structures. TiML also builds upon these facilities to support specifying complexities in the big-O asymptotic notion.

TiML's time-complexity support begins by allowing the programmer to put a number above the "arrow" of each function type (e.g. $\mathsf{int} \xrightarrow{5} \mathsf{int}$), representing an upper bound on its running time. This number is called an index of the function type. Since a function's running time often depends on the size of its input, datatypes can also contain indices representing their sizes. A function can be parametric on indices in order to accept inputs of any size (e.g. $\forall n.\ \mathsf{list\ int}\ n \xrightarrow{n^2} \mathsf{list\ int}\ n$). Inspired by DML, TiML does not fix an indexing scheme for datatypes (like length-indexed lists) but instead lets the user provide indexing schemes in the definitions of datatypes, doing away with any built-in notion of "size." This flexibility allows the programmer to choose size notions like depth of a tree, black-depth of a red-black-tree, largest element in a list, etc.

Many data structures (e.g. balanced search trees) have invariants involving their sizes, and many functions require/guarantee constraints on their input/output sizes. To make these requirements formal, TiML classifies indices with sorts and introduces a special form of sorts called refinement sorts. Sorts are to indices what types are to

terms, and refinement sorts are like refinement types [70] but on the index/sort level. A refinement sort denotes a subset of indices of the base sort satisfying a predicate (e.g. $\{n : \mathsf{Nat} \mid n \bmod 2 = 0\}$).

A syntax-directed algorithmic version of typing rules is derived for typechecking, and the refinement predicates in sorts will cause the typechecker to generate verification conditions (VCs) which are discharged by an SMT solver. With full annotations of the running time of recursive functions, the VCs are regular inequality formulas like $3(n-1)+3 \leq 3n$. If the programmer decides that coming up with a time-complexity annotation like $3n$ is too burdensome, she can choose to omit this annotation, and the inference-enabled typechecker will generate VCs like $T(n-1)+3 \leq T(n)$ with an unknown $T(\cdot)$. In this situation we face the problem of recurrence solving, which TiML handles in an incomplete way by using heuristic pattern-matching-based big-O complexity inference that can handle recurrences resulting from many common iteration and divide-and-conquer patterns[2]. For example, seeing a pattern $T(n-1)+3 \leq T(n)$, the solver infers that the function's time complexity is $O(n)$; seeing a pattern $2T(\lfloor n/2 \rfloor)+4n+5 \leq T(n)$, the solver infers that the function's time complexity is $O(n \log(n))$. Big-O bounds are expressed in TiML as sorts refined by the big-O predicate, which is a binary relation between two indices of a function sort (sorts include not only natural and real numbers but also function sorts classifying functions from indices to indices).

I formalized a core calculus of TiML in Coq and proved its soundness (i.e. well-typed programs will never go wrong, and types really bound actual running time). The formalization effort is by itself a sizable project and offered some techniques, insights, and lessons on formal reasoning about indexed/refinement type systems.

1.4.2 TiEVM: the target language

TiML is a high-level language to be used by programmers directly. The resource bounds established by its type system will only hold in the actual execution if the compiler respects the cost model used in TiML's operational semantics. To make

[2] Because it is heuristic-based, I do not have a clear characterization of its applicability, though.

sure that the resource-bound guarantees established by the TiML typechecker will be carried down to the assembly level, I designed a typed assembly language with resource bounds in a similar fashion to TiML's, so that when the assembly program passes the assembly-level typechecker, its resource use is guaranteed to be bounded by the specifications in types. I developed a type-preserving compiler to translate TiML programs into typed assembly programs.

I designed two typed assembly languages for this thesis. The first one, TiTAL (Timed TAL), is a minimalist assembly language inspired by the Typed Assembly Language (TAL) work [63]. It is a register-based language that contains a few instructions for arithmetic, heap-allocated arrays, and jumps. I used it to explore the design space of a typed assembly language with resource bounds and its compiler.

The second language, TiEVM (Timed EVM), is a much larger language that covers most parts of EVM, serving as a typed version of it. It is designed specifically to fit in the EVM machine model, with many design choices motivated by constraints imposed by EVM. Targeting a real-world assembly language incurs much more work than a minimalist toy language, but it reveals many subtleties that one needs to consider when trying to realize the benefits of static cost analyses in a real-world setting where costs depend on many machine details and design choices are constrained by machine realities, which makes this endeavor worthwhile. I admit that EVM is still a virtual machine compared to real hardware, and many issues in measuring real hardware performance, such as caching and speculative branching, do not arise in the EVM cost model. EVM serves as a "realistic enough" target machine for me, with the largest advantage being that it has an official cost model. Two versions of the compiler are implemented, targeting TiTAL and TiEVM respectively. In this thesis I will focus on TiEVM and the compiler targeting it.

The design principle of TiEVM (and generally of any typed assembly language) is that all data locations (registers, memory, etc.) and jump destinations must have types. The types associated with data locations regulate the kinds of operations that can be performed on that data (e.g. arithmetic calculations cannot be performed on strings); the types at jump destinations describe what the following code expects

of the environment (the "pre-condition"). If the type system is higher-order (i.e. contains function types), the types at jump destinations can also describe what the environment will be (the "post-condition") when the following code finishes (jumps away), by giving the type of the return pointer. This style of including a specification of the return pointer in the specification at a function entry point is called continuation-passing style (CPS) [15].

To specify how much of different resources a piece of assembly code will use, I also follow the continuation-passing style. The type at a jump destination describes how much resource the program will use from now till the very end of the execution; within this type, the type of the return pointer describes how much of the resources will be used after jumping to the return address. The delta of the two will be the resources consumed by the code snippet following the entry point (a basic block). The merit of this style will be discussed in Chapter 4.

TiEVM's types are also indexed. Its type system shares the same index and sort subsystem with TiML. There is also an overlap between TiEVM's and TiML's types, mostly the primitive ones. For compound types such as functions, tuples, and arrays, TiEVM has its own versions that are more primitive and reveal low-level details.

The counterparts of TiML expressions in TiEVM are bytecode instructions. Most instructions correspond directly to EVM instructions, but a few of them are "pseudo-instructions" absent in EVM. These pseudo-instructions are there to provide certain primitive operations to make TiEVM's semantics simpler. Each of them can be expanded into a short sequence of EVM instructions. Pseudo-instructions will be discussed in more detail in Chapter 3.

The use case of writing smart contracts forces me to extend TiML and TiEVM from a "pure mode" to a "monadic mode." More precisely, it forces me to add "strong updates," meaning updates that will change the typing context. The reason is that in a pure functional program, a function's resource costs only depend on the input arguments. But for a smart contract, its execution often depends on the current values of some data in storage. The function type needs to be extended with a precondition describing the pre-state of the storage data and a postcondition for the post-state.

1.4.3 The type-preserving compiler

The design of the compiler is inspired by [63]. Main compilation phases include CPS conversion, closure conversion, and code generation. All of these phases transform both programs and types, generating well-typed programs. Resource bounds are kept identical from the beginning to the end, so bounds specified in the source program have the same meaning and unit of measure as the actual resource usage of the generated bytecode.

There is another compilation phase at the beginning of the pipeline, translating a surface version of TiML to the core version formalized in Chapter 2. The surface version of TiML provides datatypes, compound patterns, a module system, and several other language features that have proved convenient to use in ML-like languages. A surface-level typechecker has been implemented for it; error messages seen by the programmer are all on this level.[3] Translating the surface version to the core version involves replacing datatypes and patterns with more primitive language features such as recursive types, existential types, and sum types, and combining all modules into a large single program.

An important issue in designing the compiler is its influence on costs. The only official cost model I have is the one for EVM, at the bottom of the compilation pipeline. Source programs are transformed by the compiler in complicated ways before being turned into EVM code, so how do we define and reason about costs at the source level? One way out is to give up on this problem, not checking costs when typechecking the source program, only doing the cost check on the assembly code generated by the compiler. I think this approach is antithetical to a principle of this thesis, which is that the programmer should be able to reason at the source level and not need to care about what the compiler does (knowing that the compiler will preserve the costs of their programs). I believe that once a source program passes the source typechecker, all compilation phases should generate well-typed programs without any errors. The rationale is that once the code has been changed by the tool,

[3]Error messages about transformed code will confuse users, demonstrated by the notoriously hard-to-read error messages for C++ templates.

it is out of the scope of the programmer's mental model, and the programmer will not be able to react to feedback in terms of the modified code.

Reasoning about costs should follow this principle. I designed a cost model for TiML that reflects the official EVM cost model and the transformations by the compiler. In other words, I back-propagated the cost model from the target language to the source language. The calculation of costs on the source level mirrors the compilation strategy but is expressed in a simpler way as mathematical formulas. For example, the compilation performs CPS conversion and closure conversion, so the cost of a function call on the source level depends on the number of free variables in the code after the function call (i.e. the "continuation"). Such a cost model for the source language is closely coupled with specific compilation strategies, but in order to have an accurate cost estimation on the source level, such coupling is unavoidable.

The need to keep the source cost model manageable incentivized me to keep the compiler simple, a desirable property by itself. But I do give up complex optimizations out of this concern, in order to achieve more accurate cost estimation. I could add more optimizations and retain the source cost model, as long as these optimizations are guaranteed not to increase actual cost. In that case the actual performance of the generated code is improved, while the accuracy of cost estimation is worse.

1.5 Novelties

The novelties of this thesis work include:

- **A novel use of refinement sorts for complexity analysis with invariants that balances expressivity and usability.**
- **Encoding the big-O notation formally in a type system.**
- **A rigorous type-soundness proof formalized in Coq.**
- **The first foundational approach to guarantee gas bounds of EVM bytecode formally**. In achieving this novelty, several components of my system also constitute notable contributions, such as **the first type system for**

EVM and **the first type-preserving compiler that also preserves the program's resource-usage guarantees**.

1.6 Notations

Natural number	n, m	List	$\overrightarrow{a}$
List literal	$[a_1, \cdots, a_n]$	List comprehension	$\{a_n \vert P(a_n)\}$
List append and concatenation	$a;l, l;a, l_1;l_2$	List length	$\vert l\vert$
Map literal	$\{k_1 \mapsto v_1, \cdots, k_n \mapsto v_n\}$	Map update	$m[k \mapsto v]$
Map comprehension	$\{k \mapsto v \vert P(k, v)\}$	List or map get-at	$l(n), m(x)$
Substitution	$b[v/x]$	Map domain	$\mathsf{dom}(m)$
Empty list or map	$\cdot$	String	u
Boolean	b		

Figure 1-1: Notations

Notations used throughout this thesis are listed in Figure 1-1. List appending $a;l$ or $l;a$ appends an element a at the front or end of the list l. List (or map) comprehension constructs a list (or map) from elements satisfying a predicate[4]. Map update is used for both adding a new key-value pair and updating the value at an existing key. I sometimes use $m; x : \tau$ to mean $m[x \mapsto \tau]$. Substitution $b[v/x]$ stands for replacing every appearance of variable x with v in b, performing alpha-conversions when necessary to avoid name capture. In this thesis I will use different letter sets to distinguish different syntax classes. For example, n, m will be used for natural numbers while i, j are for "indices" (defined in Chapter 2).[5]

1.7 Source code

The source code of the whole system and the example programs used in evaluation can be found at `https://github.com/wangpengmit/phd-thesis-supplemental`.

[4]List comprehension preserves the ordering of the original list $[a_1, \cdots, a_n]$.

[5]I agree that this is an unfortunate way to distinguish mathematical objects of different kinds. In a functional programming language like SML or Coq they are easily distinguished by different types or datatype constructors, but mathematical notations are an ancient system that has not been catching up with modern functional-programming practices.

Chapter 2

TiML

.

This chapter describes TiML, a typed functional programming language with resource bounds and the source language of the type-preserving compiler. I will first demonstrate TiML by showing some code examples and then formally define the TiML language by describing its grammar, operational semantics, and typing rules. After that I will discuss the proof of the type system's soundness formalized in the Coq proof assistant. In the last section, I will describe a variant of TiML called ETiML that is designed for writing smart contracts on the Ethereum platform, with features and modifications purposefully chosen for this setting.

2.1 TiML examples

```
datatype list α : {ℕ} = Nil of list α {0}
                      | Cons {n : ℕ} of α * list α {n}⟶list α {n + 1}

fun foldl [α β] {m n : ℕ} (f : α * β -[$m]→ β) acc (l : list α {n})
    return β using $(m + 4) * $n =
  case l of
      [] ⇒ acc
    | x :: xs ⇒ foldl f (f (x, acc)) xs
```

Figure 2-1: TiML example: definition of list and fold-left

In this section, I will give a tutorial introduction to TiML programming. The first TiML example is the fold-left function on lists, through which I introduce the basics of the language. Figure 2-1 lists the definition of list and fold-left. TiML mimics the syntax of Standard ML (SML) [2], on top of which I add indices and sorts. Datatype `list` is parametrized not only on a type variable α but also on an index of sort $\mathbb{N}$ (natural numbers), expressed by the "`: {`$\mathbb{N}$`}`" part in the header. This index argument varies in different constructors. In the `Nil` case, it is fixed to 0, while for `Cons`, it is `n+1`, where `n` is the index of the tail list. Obviously this index represents the length of a list. Note that the TiML language itself has no built-in knowledge of "sizes"; they are just a datatype's index arguments, which may happen to correspond to some human intuition about how big a chunk of data is[1].

The fold-left function `foldl` takes two type arguments (unlike SML they are explicit in TiML) α and β, two index arguments `m` and `n`, an operation `f` to be performed on the list, an accumulator `acc`, and an input list `l`, and returns a result of type β. The two indices stand for the time bound of operation `f` and the length of the input list. The sort `Time` is defined as nonnegative real numbers; the `$` sign is used to convert a natural number to a time. The real number domain is chosen because TiML's time expressions allow logarithms. Arrows in function types are extended to "long arrows" (e.g. $-\boxed{\$m}\rightarrow$) carrying time specifications on their shoulders. Recursive functions need to be annotated with return types and time bounds via the `return` and `using` keywords.

Cost model: TiML can work with different cost models by choosing the cost parameters in its typing rules (as will be shown in Figure 2-10). In this chapter (except Section 2.6), for illustration and simplification purpose, I use a simple cost model where all memory costs are ignored (i.e. memory costs are always zero) and only a function application costs one unit of time. All other operations (including constructor applications) do not consume time. From Section 2.6 on, I will switch to a realistic cost model that matches the official EVM specification. Section 4.5

[1]The meta-theoretic treatment does not talk about "sizes," and the user does not get formal guarantees about "sizes" from the soundness theorem in Section 2.5, so "sizes" can be regarded as a proof intermediary for "time."

discusses cost models in more detail.

Here, `foldl`'s running time is bounded by `$(m + 4) * $n` ("4" comes from these four function applications illustrated by the dots: `foldl . f . (f . (x, acc)) . xs`).

Typechecking `foldl` will generate one verification condition (VC):

$$\forall m\ n\ n' : \mathsf{Nat}.\ n' + 1 = n \rightarrow m + 4 + (m+4)n' \leq (m+4)n.$$

n' is introduced by constructor `Cons` to represent the length of the tail list, and the premise $n' + 1 = n$ is introduced by typechecking the pattern-matching, which connects the inner index n' to the outer index argument n. The inequality $m+4+(m+4)n' \leq (m+4)n$ dictates that the actual running time of this branch, $m+4+(m+4)n'$, should be bounded by the specified bound $(m+4)n$.

The next example is merge sort (Figure 2-2), in which I show the use of the big-O notation to reduce annotation burden. Let us first look at the function `msort`. Instead of using a concrete time bound such as `$(m + 4) * $n`, I bound `msort`'s running time by "`T_msort m n`". `T_msort` is an index of sort "`BigO (λ m n ⇒ $m * $n * log₂ $n)`", which is the sort of *time functions* (functions from multiple natural numbers to time) that are in the big-O class $O(mn \log_2 n)$. Under the hood, "`BigO f`" is syntax sugar for { `g | g ≦ f` }, the sort of time functions refined by the big-O binary relation ≦. Formally, this multivariate big-O relation is defined axiomatically according to [51], meaning that any relation that satisfies the five axioms in [51] can be used as the big-O relation here. [51] also gives an instance that satisfies these axioms. TiML's big-O inference engine only relies on these five axioms.

The point of using big-O to specify `T_msort` is that only the sort of `T_msort` is needed, not the definition (written as an underscore). The typechecker generates VC

$$T \leqq (\lambda m\, n.\ mn \log_2 n) \wedge \forall m\ n.\ T(m, \lceil n/2 \rceil) + T(m, \lfloor n/2 \rfloor) + 7 + T_{\mathsf{split}} + T_{\mathsf{merge}} \leq T(m,n), \quad (2.1.0.1)$$

for which $T_{\mathsf{split}} \leqq (\lambda n.\ n)$ and $T_{\mathsf{merge}} \leqq (\lambda m\ n.\ m \times n)$ are available premises in

```
absidx T_split: BigO _ (* (fn n => $n) *) = _
fun split [α] {n: ℕ} (l: list α {n})
    return list α {ceil ($n/2)} * list α {floor ($n/2)} using T_split n =
  case l of
      [] ⇒ ([], [])
    | [_] ⇒ (l, [])
    | x1 :: x2 :: xs ⇒
      let val (xs1, xs2) = split xs in
        (x1 :: xs1, x2 :: xs2) end

absidx T_merge: BigO (λ m n ⇒ $m * $n) = _
fun merge [α] {m n1 n2: ℕ} (le: α * α -[$m]→ bool)
      (xs: list α {n1}, ys: list α {n2})
    return list α {n1 + n2} using T_merge m (n1 + n2) =
  case (xs, ys) of
      ([], _) ⇒ ys
    | (_, []) ⇒ xs
    | (x :: xs', y :: ys') ⇒
      if le (x, y) then x :: merge le (xs', ys)
      else y :: merge le (xs, ys')

absidx T_msort: BigO (λ m n ⇒ $m * $n * log2 $n) = _
fun msort [α] {m n: ℕ} (le: α * α -[$m]→ bool) (xs: list α {n})
    return list α {n} using T_msort m n =
  case xs of
    [] ⇒ xs
  | [_] ⇒ xs
  | _ :: _ :: _ ⇒
    let val (xs1, xs2) = split xs in
    merge le (msort le xs1, msort le xs2) end
```

Figure 2-2: TiML example: merge sort

the context, and T is an unknown variable. This VC is discharged by the heuristic pattern-matching-based recurrence solver. The solver will not give a solution for T; instead, it sees that this VC matches the pattern of one of the cases in the Master Theorem [27], therefore concluding that such a T exists and this VC is true. The absence of a concrete definition of T (i.e. `T_msort`) is OK because `T_msort` is declared as an *abstract index* by `absidx`. Outside the current module, its definition is not visible.

Since the Master Theorem can decide the solution's big-O class from the recurrence, the big-O class in a big-O sort can also be omitted and inferred, as shown by function `split`. Notice that its big-O class `(λ n ⇒ $n)` is commented out.

```
datatype color : {𝔹} =
  Black of color {true}
| Red of color {false}

datatype rbt a : {ℕ} {𝔹} {ℕ} =
  Leaf of rbt a {0} {true} {0}
| Node {lcolor color rcolor : 𝔹}
       {lsize rsize bh (*black-height*) : ℕ}
       {color = false → lcolor = true ∧ rcolor = true}
       { ... (*other invariants*)}
  of color {color} * rbt a {lsize} {lcolor} {bh} * (key * a)
       * rbt a {rsize} {rcolor} {bh}
      ⟶rbt a {lsize + 1 + rsize} {color} {bh + b2n color}
```

Figure 2-3: TiML example: red-black trees

The last example is the definition of red-black trees (Figure 2-3), where I need to encode the invariants of the data structure. A red-black tree `rbt` is indexed by three indices: the size, the root color, and the black-height. I use { `P` } as syntax sugar for { `_ :` { `_ : unit | P` } } where the index itself is not interesting. `Leaf` is black with zero size and zero black-height. In the case of `Node`, the children must have the same black-height, and when the root color is red, the children's root colors must both be black. `b2n` does conversion from Booleans `true` and `false` to natural numbers `1` and `0` respectively.

2.2 Syntax and semantics

The TiML code above uses the surface syntax understood by the parser. In this section I define TiML as a formal calculus whose soundness I prove. The translation from the surface language to the formal calculus will be described in Section 4.1 as a phase of the compilation pipeline.

2.2.1 Syntax

$$
\begin{array}{lll}
\text{Base Sort} & & \\
\quad \underline{s} & ::= & \mathsf{Nat} \mid \mathsf{Time} \mid \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{State} \mid \underline{s} \Rightarrow \underline{s} \\
\text{Sort} & & \\
\quad s & ::= & \underline{s} \mid \{a : \underline{s} \mid \theta\} \\
\text{Index} & & \\
\quad i & ::= & a \mid n \mid r \mid () \mid \mathsf{true} \mid \mathsf{false} \mid o_{ui}\ i \mid i\ o_{bi}\ i \mid i\ ?\ i : i \mid \lambda a : \underline{s}.\ i \mid i\ i \mid \{\overrightarrow{u : i}\} \\
\text{Proposition} & & \\
\quad \theta & ::= & b \mid \neg\theta \mid \theta \wedge \theta \mid \theta \vee \theta \mid \theta \rightarrow \theta \mid i\ o_{br}\ i \mid \forall a : \underline{s}.\ \theta \mid \exists a : \underline{s}.\ \theta \\
\text{Kind} & & \\
\quad \kappa & ::= & * \mid \kappa \Rightarrow \kappa \mid \underline{s} \Rightarrow \kappa \\
\text{Type} & & \\
\quad \tau & ::= & \alpha \mid \mathsf{unit} \mid \mathsf{byte} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{nat}\ i \mid \mathsf{ibool}\ i \mid \mathsf{array}_n\ \tau\ i \mid \tau \xrightarrow{i} \tau \mid \mathsf{tuple}\ \overrightarrow{\tau} \mid \mathsf{record}\ \{\overrightarrow{u : \tau}\} \\
 & \mid & \tau + \tau \mid \mu\alpha : \kappa.\ \tau \mid \forall^{i}_{\alpha:\kappa}.\ \tau \mid \forall^{i}_{a:s}.\ \tau \mid \exists\alpha : \kappa.\ \tau \mid \exists a : s.\ \tau \mid \lambda\alpha : \kappa.\ \tau \mid \tau\ \tau \mid \lambda a : \underline{s}.\ \tau \mid \tau\ i \\
 & \mid & \mathsf{state}\ u \mid \mathsf{map}\ \tau \mid \mathsf{vector}\ \tau \mid \mathsf{cell}\ \tau \mid \mathsf{icell}\ \tau \mid \mathsf{ptr}\ \tau \\
\text{Term} & & \\
\quad e & ::= & x \mid () \mid n \mid \overline{n} \mid \mathsf{byte}\ n \mid \lambda x : \tau.\ e \mid e\ e \mid o_{\mathsf{ut}}\ e \mid e\ o_{\mathsf{bt}}\ e \mid e \oplus e \mid (e, e) \mid e.1 \mid e.2 \mid \mathsf{l}_\tau.e \mid \mathsf{r}_\tau.e \\
 & \mid & \mathsf{case}\ e\ \mathsf{of}\ x.e\ \mathsf{or}\ x.e \mid \mathsf{fold}_\tau\ e \mid \mathsf{unfold}\ e \mid \Lambda\alpha : \kappa.\ e \mid e\ \tau \mid \Lambda a : s.\ e \mid e\ i \mid \mathsf{pack}_\tau\ \langle\tau, e\rangle \\
 & \mid & \mathsf{unpack}\ e\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ e \mid \mathsf{pack}_\tau\ \langle i, e\rangle \mid \mathsf{unpack}\ e\ \mathsf{as}\ \langle a, x\rangle\ \mathsf{in}\ e \mid \mathsf{rec}_\tau\ x.\ e \mid \mathsf{new}_n\ e\ e \\
 & \mid & \mathsf{arrayFromList}_n\ \{\overrightarrow{e}\} \mid \mathsf{read}_n\ e\ e \mid \mathsf{write}_n\ e\ e\ e \mid \mathsf{len}\ e \mid \ell \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
 & \mid & b \mid \overline{b} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{ifi}\ e\ \mathsf{then}\ x.e\ \mathsf{else}\ x.e
\end{array}
$$

Figure 2-4: TiML syntax

The syntax of TiML is given in Figure 2-4. I split the core language into four syntactic classes: terms, types, indices, and sorts. Terms are classified by types; types can be indexed by indices, which are classified by sorts[2]. In this dissertation I will use the words “term” and “expression” interchangeably.

[2]Alternatively one can treat both types and indices as type-level constructors and sorts as kinds. We keep types and indices separate because they are treated differently in our metatheoretical development (e.g. indices are given a denotational semantics while types are not).

Unary Index Operator			Binary Index Relation		
o_{ui}	::=	ceil | floor | neg | div_n | $\log_n$	o_{br}	::=	$=$ | $\leq$ | $<$ | $\geq$ | $>$ | $\leqq$
	|	exp_n | nat2time | bool2nat			
Binary Index Operator			**Binary Term Operator**		
o_{bi}	::=	$+$ | $-$ | $\times$ | max | min | and	o_{bt}	::=	$+$ | $-$ | $\times$ | $/$ | $\cup$ | $\cdots$
	|	or | $=?$ | $\leq?$ | $<?$ | $\geq?$ | $>?$			

Figure 2-5: Operators

TiML's base sorts include natural numbers, time (nonnegative real numbers), unit, Booleans, states, and functions from base sorts to base sorts. A sort is either a base sort or a base sort refined by a proposition. In refinement sort $\{a : \underline{s} \mid \theta\}$, the variable a stands for the index being refined and can be mentioned by the proposition θ. An index, ranged over by i or j, can be an index variable a, a constant natural number n, a constant nonnegative real number r, the unit value $()$, or a Boolean constant. It can also be formed by unary index operation $o_{ui}\ i$, binary index operation $i\ o_{bi}\ i$, if-then-else, lambda abstraction, application, or state specification. The last index form, of sort State, is used to express state specifications that will be described in Section 2.6. We use different letter sets to denote different kinds of variables: a and b for index variables, α and β for type variables, and x and y for term variables. All operators are summarized in Figure 2-5. Propositions include usual logical constructs and binary relations $i\ o_{br}\ i$ between two indices. There is a special binary relation "$f \leqq g$" between two time functions meaning $f \in O(g)$.

TiML has a higher-order kind system with the $\kappa \Rightarrow \kappa$ kind-former, besides which there is also the kind-former $\underline{s} \Rightarrow \kappa$, allowing type-level functions from base sorts to types, which is needed for kind-indexed datatypes like `list` in Figure 2-1. TiML's base types include unit, bytes, integers (512-bit)[3], Booleans, indexed natural numbers, and indexed Booleans. Integers are just an example of ordinary primitive types. It has length-indexed arrays, which are a generalization of mutable references. The length index is useful in complexity analysis of array-based algorithms, and it also brings the benefit of static array-bound checking. Arrays are parametrized by a subscript n which is the width of its elements (in units of bytes). This parameter cannot always

[3]See Chapter 3 for why the word width is 512 bit.

be implied by the element type because the type may be a type variable[4]. Arrays with different element widths are considered different types.

Indexed natural numbers serve as a bridge between runtime values and static indices. For example, the only value of type $\mathsf{nat}\ 3$ is the term $\overline{3}$. Without indexed natural numbers, all integers would just have type int, and at compile time it would be impossible to know their runtime values from their types, despite the fact that e.g. array-bound checking is done entirely based on types[5]. Indexed natural numbers give one the opportunity to know a variable's runtime value from its type. Indexed Booleans play a similar role.

The function type (arrow type) is indexed by an upper bound on the function's running time, which is the most crucial new feature in TiML. Tuple/sum/recursive types are supported to enable user-defined datatypes. Tuples must have at least two components. I sometimes write a tuple type as a product type $\tau_1 \times \cdots \times \tau_n$, and sometimes only discuss the $\tau_1 \times \tau_2$ when generalizing from pairs to tuples is straightforward. Records are turned into tuples internally so I will not discuss records in this thesis.

For polymorphic types, existential types, type-level abstractions, and type-level applications, I have two versions of each, one for type arguments and one for index arguments. I use the same notation in this document for the two versions (and the corresponding introduction and elimination term forms), relying on context to tell them apart. Polymorphic types ($\forall$ types) are also indexed by a cost bound, akin to arrow types, because in some cost models an index/type abstraction also needs some computation to produce a result when applied. In the simple cost model used in this chapter, this bound is always zero and often omitted. The last few types from $\mathsf{state}\ u$ are for the state mechanism, which will be described in Section 2.6 and Chapter 3.

Now let us turn to terms (expressions). Some terms are annotated with types (shown as subscripts) to facilitate syntax-directed typechecking, and many of these annotations can be inferred or disguised as datatypes. I use n and $\overline{n}$ as distinct

[4] TiML's kind system does not contain width information for a type variable.
[5] TiML is not dependently typed.

syntax classes to represent integer constants and natural-number constants, of types int and $\mathsf{nat}\ i$ respectively. $e\ o_{\mathsf{bt}}\ e$ stands for all primitive binary operations working on primitive types such as int. $e \oplus e$ is the plus operation on natural numbers[6]. I use addition as an example for other possible natural-number operations. The next few term formers are the introduction and elimination forms of the corresponding types. I use the notation $x.e$ to mean a binding where x is locally bound in e. $\mathsf{rec}_\tau\ x.\ e$ is a general fixpoint.

The next few terms are for array creation, read, write, and length retrieval[7]. $\mathsf{new}\ e_1\ e_2$ is for creating an array of size e_1 with all elements initialized to be e_2. $\mathsf{arrayFromList}_n\ \{\overrightarrow{e}\}$ is for creating an array whose elements are $e_1, \cdots, e_n$. Locations ℓ only arise during reduction, as in standard operational semantics for mutable references. $\mathsf{ifi}\ e\ \mathsf{then}\ x.e_1\ \mathsf{else}\ x.e_2$ is for indexed Booleans what if-then-else is for Booleans. The variable x in each branch contains a proof that e evaluates to $\overline{\mathsf{true}}$ (or $\overline{\mathsf{false}}$) (see its typing rule in Figure 2-10).

2.2.2 Operational semantics

TiML's operational semantics (Figure 2-6 and 2-7) is a standard small-step operational semantics instrumented with a "fuel" parameter. Fuel (a nonnegative real number) is consumed by certain reductions, and a reduction without enough fuel to proceed is stuck. A starting fuel amount in an execution that does not get stuck will thus be an upper bound on the execution's total time cost (assuming fuel consumption coincides with time consumption). Formally, reductions are defined between configurations, each a triple of a heap, a program, and a fuel amount. A heap is a finite partial map (denoted as $\rightharpoonup$) from locations to lists of values (denoted as $\overrightarrow{v}$). Note that each location points to a list of values instead of a single value because each location corresponds to an array. The reduction relation $\sigma \rightsquigarrow \sigma'$ is defined via

[6]It is different from $e\ o_{\mathsf{bt}}\ e$ because the result type has to be properly indexed to reflect the computation. For example, supposing $\vdash e_1 : \mathsf{nat}\ 3$ and $\vdash e_2 : \mathsf{nat}\ 4$, then $\vdash e_1 \oplus e_2 : \mathsf{nat}\ 7$. Notice that $(e_1 \oplus e_2)$'s type is indexed, and the index is computed from the indices of the operands' types. On the contrary, if $\vdash e_1 : \mathsf{int}$ and $\vdash e_2 : \mathsf{int}$, then $\vdash e_1\ o_{\mathsf{bt}}\ e_2 : \mathsf{int}$ (where o_{bt} is e.g. plus), an unindexed type.

[7]array creation/read/write are all annotated with the element width n.

Value

$$
\begin{array}{rcl}
v & ::= & () \mid d \mid \overline{n} \mid \lambda x:\tau.\ e \mid (v,v) \mid \mathsf{l}_\tau.v \mid \mathsf{r}_\tau.v \mid \mathsf{fold}_\tau\ v \mid \Lambda\alpha:\kappa.\ e \mid \Lambda a:s.\ e \mid \mathsf{pack}_\tau\ \langle\tau,v\rangle \\
 & \mid & \mathsf{pack}_\tau\ \langle i,v\rangle \mid \ell
\end{array}
$$

Evaluation Context

$$
\begin{array}{rcl}
E & ::= & \square \mid E\ e \mid v\ E \mid E\ o_{bt}\ e \mid v\ o_{bt}\ E \mid E \oplus e \mid v \oplus E \mid (E,e) \mid (v,E) \mid E.1 \mid E.2 \mid \mathsf{l}_\tau.E \mid \mathsf{r}_\tau.E \\
 & \mid & \mathsf{case}\ E\ \mathsf{of}\ x.e\ \mathsf{or}\ x.e \mid \mathsf{fold}_\tau\ E \mid \mathsf{unfold}\ E \mid E\ \tau \mid E\ i \mid \mathsf{pack}_\tau\ \langle\tau,E\rangle \mid \mathsf{unpack}\ E\ \mathsf{as}\ \langle\alpha,x\rangle\ \mathsf{in}\ e \\
 & \mid & \mathsf{pack}_\tau\ \langle i,E\rangle \mid \mathsf{unpack}\ E\ \mathsf{as}\ \langle a,x\rangle\ \mathsf{in}\ e \mid \mathsf{new}\ E\ e \mid \mathsf{new}\ v\ E \mid \mathsf{read}\ E\ e \mid \mathsf{read}\ v\ E \\
 & \mid & \mathsf{write}\ E\ e\ e \mid \mathsf{write}\ v\ E\ e \mid \mathsf{write}\ v\ v\ E \mid \cdots
\end{array}
$$

Heap

$$
h \in \mathsf{loc} \rightharpoonup \overrightarrow{v}
$$

Configuration

$$
\sigma = (h,e,r)
$$

Figure 2-6: Definitions in operational semantics

evaluation contexts and the atomic reduction relation $\sigma \twoheadrightarrow \sigma'$. In the current setting, only the beta reduction (function application) consumes fuel, by 1.

Fixpoint unrolling does not consume fuel, which appears to make the counting scheme too lax, as one may suspect that it is counting too few steps, where fixpoint unrollings can cause unbounded reduction sequences on their own. However, the typing rules induce a syntactic restriction ruling out two consecutive fixpoint unrollings without a beta reduction (explained more later).

Array reads and writes check that the offset is within bounds before performing the operation. I use notation $m[k \mapsto v]$ to mean updating a map or a list at k and v^n to mean a list of v repeated n times. $[\![o_{\mathsf{bt}}]\!]$ is an interpretation of the primitive binary operation, which is a partial function that can result in an error when given illegal arguments. For symmetric pairs like $e.1$ and $e.2$, we only show one.

2.3 Type system

TiML's type system consists of various forms of judgments including sorting, kinding, typing, type equivalence, wellformedness of various entities, etc. Here I only describe sorting and typing rules (in Figure 2-9 and 2-10), since they are most relevant to complexity analysis. The rules use several kinds of contexts, which are summarized

$\boxed{\sigma \rightharpoonup \sigma'}$

$$\frac{r \geq 1}{(h, (\lambda x : \tau.\ e)\ v, r) \rightharpoonup (h, e[v/x], r-1)} \qquad \frac{}{(h, \mathsf{rec}_\tau\ x.\ e, r) \rightharpoonup (h, e[\mathsf{rec}_\tau\ x.\ e/x], r)}$$

$$\frac{}{(h, \mathsf{unpack}\ (\mathsf{pack}_{\tau'}\ \langle \tau, v\rangle)\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ e, r) \rightharpoonup (h, e[\tau/\alpha][v/x], r)}$$

$$\frac{}{(h, \mathsf{unpack}\ (\mathsf{pack}_{\tau}\ \langle i, v\rangle)\ \mathsf{as}\ \langle a, x\rangle\ \mathsf{in}\ e, r) \rightharpoonup (h, e[i/a][v/x], r)}$$

$$\frac{}{(h, (\Lambda\alpha : \kappa.\ e)\ \tau, r) \rightharpoonup (h, e[\tau/\alpha], r)} \qquad \frac{}{(h, (\Lambda a : s.\ e)\ i, r) \rightharpoonup (h, e[i/a], r)} \qquad \frac{}{(h, (v_1, v_2).1, r) \rightharpoonup (h, v_1, r)}$$

$$\frac{}{(h, \mathsf{case}\ \mathsf{l}_\tau.v\ \mathsf{of}\ x.e_1\ \mathsf{or}\ x.e_2, r) \rightharpoonup (h, e_1[v/x], r)} \qquad \frac{h(\ell) = \overrightarrow{v} \qquad n < |\overrightarrow{v}|}{(h, \ell[\overline{n}], r) \rightharpoonup (h, v_n, r)}$$

$$\frac{h(\ell) = \overrightarrow{v} \qquad n < |\overrightarrow{v}|}{(h, \ell[\overline{n}] := v', r) \rightharpoonup (h[\ell \mapsto \overrightarrow{v}[n \mapsto v']], (), r)} \qquad \frac{\ell \notin h}{(h, \mathsf{new}\ \overline{n}\ v, r) \rightharpoonup (h[\ell \mapsto v^n], \ell, r)}$$

$$\frac{[\![o_{\mathsf{bt}}]\!](v_1, v_2) = v}{(h, v_1\ o_{\mathsf{bt}}\ v_2, r) \rightharpoonup (h, v, r)} \qquad \frac{}{(h, \overline{n_1} \oplus \overline{n_2}, r) \rightharpoonup (h, \overline{n_1 + n_2}, r)} \qquad \frac{}{(h, \mathsf{unfold}\ (\mathsf{fold}_\tau\ v), r) \rightharpoonup (h, v, r)}$$

$\boxed{\sigma \rightsquigarrow \sigma'}$

$$\frac{(h, e, r) \rightharpoonup (h', e', r')}{(h, E[e], r) \rightsquigarrow (h', E[e'], r')}$$

Figure 2-7: Operational semantics

Sorting Context	Kinding Context	Typing Context
$\Omega \ :\ \{\overrightarrow{a \mapsto s}\}$	$\Lambda \ :\ \{\overrightarrow{\alpha \mapsto \kappa}\}$	$\Gamma \ :\ \{\overrightarrow{x \mapsto \tau}\}$
Heap Type	Full Context	
$\Sigma \ :\ \{\overrightarrow{l \mapsto (\tau, i)}\}$	$\Delta \ =\ (\Omega, \Lambda, \Gamma, \Sigma)$	

Figure 2-8: Typing contexts

$\boxed{\Omega \vdash i : s}$

$$\frac{\Omega(a) = s}{\Omega \vdash a : s}\ \textsc{Var} \qquad \frac{}{\Omega \vdash n : \mathsf{Nat}}\ \textsc{Nat} \qquad \frac{}{\Omega \vdash r : \mathsf{Time}}\ \textsc{Time} \qquad \frac{}{\Omega \vdash () : \mathsf{Unit}}\ () \qquad \frac{}{\Omega \vdash \mathsf{true} : \mathsf{Bool}}\ \textsc{True}$$

$$\frac{}{\Omega \vdash \mathsf{false} : \mathsf{Bool}}\ \textsc{False} \qquad \frac{\Omega \vdash i : o_{\mathsf{ui}}.\underline{s}_1}{\Omega \vdash o_{\mathsf{ui}}\ i : o_{\mathsf{ui}}.\underline{s}_{\mathsf{res}}}\ \mathrm{UO} \qquad \frac{\Omega \vdash i_m : o_{\mathsf{bi}}.\underline{s}_m\ (m = 1, 2)}{\Omega \vdash i_1\ o_{\mathsf{bi}}\ i_2 : o_{\mathsf{bi}}.\underline{s}_{\mathsf{res}}}\ \mathrm{BO}$$

$$\frac{\Omega; a : \underline{s}_1 \vdash i : \underline{s}_2}{\Omega \vdash \lambda a : \underline{s}_1.\ i : \underline{s}_1 \Rightarrow \underline{s}_2}\ {\Rightarrow}\mathrm{I} \qquad \frac{\Omega \vdash i_1 : \underline{s}_1 \Rightarrow \underline{s}_2 \qquad \Omega \vdash i_2 : \underline{s}_1}{\Omega \vdash i_1\ i_2 : \underline{s}_2}\ {\Rightarrow}\mathrm{E} \qquad \frac{\Omega \vdash i : \underline{s} \qquad \Omega \vdash \theta[i/a]}{\Omega \vdash i : \{a : \underline{s} \mid \theta\}}\ \{\}\mathrm{I}$$

$$\frac{\Omega \vdash i : \{a : \underline{s} \mid \theta\} \qquad \Omega; a : \underline{s} \vdash \mathsf{wf}\ \theta}{\Omega \vdash i : \underline{s}}\ \{\}\mathrm{E} \qquad \frac{\Omega \vdash i : \mathsf{Bool} \qquad \Omega \vdash i_m : s\ (m = 1, 2)}{\Omega \vdash i\ ?\ i_1 : i_2 : s}\ \textsc{Ite}$$

Figure 2-9: Sorting rules

in Figure 2-8. A sorting/kinding/typing context maps index/type/term variables to sorts/kinds/types respectively. A heap typing context maps locations to type-index pairs specifying the types and lengths of arrays. A full context (used in typing rules) consists of all the above contexts. Some judgments (e.g. sorting) do not need the full context. To reduce notational noise, I still pass a full context to those judgments and elide the selection of needed parts. Similarly I will write $\Delta; x : \tau$ and $\Delta(x)$ when it is clear which component of Δ is operated on. When I want to be specific, I use e.g. $\Delta.1$ to mean the first component of Δ.

2.3.1 Typing rules

In Figure 2-9, the most important rules are the introduction and elimination rules for refinement sorts: rules {}I and {}E. An inhabitant of a base sort can be admitted by a refinement sort if it satisfies the refinement predicate. I use the validity judgment $\Omega \vdash \theta$ to mean that proposition θ is true under the context Ω, which may contain refinements to be used as premises. Its definition is sketched in Section 2.5. A member of a refinement sort can automatically be used as a member of the base sort. A subsorting rule can be derived where subsorting is defined by implication between refinement predicates. Operators can have associated information like the type of the

$$\boxed{\Delta \vdash e : \tau \rhd i}$$

$$\frac{\Delta(x) = \tau}{\Delta \vdash x : \tau \rhd 0}\ \textsc{Var} \qquad \frac{\Delta \vdash e_1 : \tau_1 \rhd i_1 \qquad \Delta; x : \tau_1 \vdash e_2 : \tau_2 \rhd i_2}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \rhd i_1 + i_2}\ \textsc{Let}$$

$$\frac{\Delta \vdash e_1 : \tau_1 \xrightarrow{i} \tau_2 \rhd i_1 \qquad \Delta \vdash e_2 : \tau_1 \rhd i_2}{\Delta \vdash e_1\ e_2 : \tau_2 \rhd i_1 + i_2 + i + C_{\mathsf{App}}(n_k, b_k)} \to\!\mathrm{E} \qquad \frac{\Delta \vdash \tau_1 :: * \qquad \Delta; x : \tau_1 \vdash e : \tau_2 \rhd i}{\Delta \vdash \lambda x : \tau_1.\ e : \tau_1 \xrightarrow{i + C_{\mathsf{AbsInner}}(e)} \tau_2 \rhd C_{\mathsf{Abs}}(e)} \to\!\mathrm{I}$$

$$\frac{e = \Lambda \overrightarrow{a : s}.\ \lambda y : \tau_1.\ e_1 \qquad \Delta \vdash \tau :: * \qquad \Delta; x : \tau \vdash e : \tau \rhd i}{\Delta \vdash \mathsf{rec}_\tau\ x.\ e : \tau \rhd i}\ \textsc{Rec}$$

$$\frac{}{\Delta \vdash () : \mathsf{unit} \rhd C_{\mathsf{Const}}}\ \textsc{Unit} \qquad \frac{}{\Delta \vdash d : \mathsf{int} \rhd C_{\mathsf{Const}}}\ \textsc{Int} \qquad \frac{}{\Delta \vdash \overline{n} : \mathsf{nat}\ n \rhd C_{\mathsf{Const}}}\ \textsc{Nat}$$

$$\frac{\Delta \vdash e_m : \tau_m \rhd i_m\ (m = 1, \cdots, n)}{\Delta \vdash (e_1, \cdots, e_n) : \tau_1 \times \cdots \times \tau_n \rhd i_1 + \cdots + i_n + C_{\mathsf{Tuple}}(n)} \times\mathrm{I} \qquad \frac{\Delta \vdash e : \tau_1 \times \cdots \times \tau_n \rhd i \qquad m \le n}{\Delta \vdash e.m : \tau_m \rhd i + C_{\mathsf{Proj}}} \times\mathrm{E}_1$$

$$\frac{\Delta \vdash e : \tau_1 \rhd i \qquad \Delta \vdash \tau_2 :: *}{\Delta \vdash \mathsf{l}_{\tau_2}.e : \tau_1 + \tau_2 \rhd i + C_{\mathsf{Inj}}} +\mathrm{I}_1$$

$$\frac{\Delta \vdash e : \tau_1 + \tau_2 \rhd i \qquad \Delta; x : \tau_m \vdash e_m : \tau \rhd i_m\ (m = 1, 2)}{\Delta \vdash \mathsf{case}\ e\ \mathsf{of}\ x.e_1\ \mathsf{or}\ x.e_2 : \tau \rhd i + C_{\mathsf{Case}}(i_1, i_2, e_1, e_2, n_k, b_k)} +\mathrm{E}$$

$$\frac{\Delta \vdash e : \mathsf{bool} \rhd i \qquad \Delta \vdash e_m : \tau \rhd i_m\ (m = 1, 2)}{\Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \rhd i + C_{\mathsf{If}}(i_1, i_2, e_1, e_2, n_k, b_k)}\ \textsc{If}$$

$$\frac{\begin{array}{c}\Delta \vdash e : \mathsf{ibool}\ j \rhd i \\ \Delta; x : \exists\{j = \mathsf{true}\}.\ \mathsf{unit} \vdash e_1 : \tau \rhd i_1 \qquad \Delta; x : \exists\{j = \mathsf{false}\}.\ \mathsf{unit} \vdash e_2 : \tau \rhd i_2\end{array}}{\Delta \vdash \mathsf{ifi}\ e\ \mathsf{then}\ x.e_1\ \mathsf{else}\ x.e_2 : \tau \rhd i + C_{\mathsf{Ifi}}(i_1, i_2, e_1, e_2, n_k, b_k)}\ \textsc{Ifi}$$

$$\frac{\Delta \vdash \tau\ \overrightarrow{c} :: * \qquad \tau = \mu\alpha :: \kappa.\ \tau_1 \qquad \Delta \vdash e : \tau_1[\tau/\alpha]\ \overrightarrow{c} \rhd i}{\Delta \vdash \mathsf{fold}_{\tau\ \overrightarrow{c}}\ e : \tau\ \overrightarrow{c} \rhd i + C_{\mathsf{Fold}}}\ \mu\mathrm{I} \qquad \frac{\tau = \mu\alpha :: \kappa.\ \tau_1 \qquad \Delta \vdash e : \tau\ \overrightarrow{c} \rhd i}{\Delta \vdash \mathsf{unfold}\ e : \tau_1[\tau/\alpha]\ \overrightarrow{c} \rhd i + C_{\mathsf{Unfold}}}\ \mu\mathrm{E}$$

$$\frac{\Delta \vdash e : \forall^i_{\alpha:\kappa}.\ \tau \rhd i_1 \qquad \Delta \vdash \tau_1 :: \kappa}{\Delta \vdash e\ \tau_1 : \tau[\tau_1/\alpha] \rhd i_1 + i + C_{\mathsf{AppT}}(n_k, b_k)}\ \forall\mathrm{E} \qquad \frac{e\ \text{is value} \qquad \Delta; \alpha :: \kappa \vdash e : \tau \rhd i}{\Delta \vdash \Lambda\alpha : \kappa.\ e : \forall^{C_{\mathsf{AbsTInner}}(e)}_{\alpha:\kappa}.\ \tau \rhd C_{\mathsf{AbsT}}(i, e)}\ \forall\mathrm{I}$$

$$\frac{\Delta \vdash e : \forall^j_{a:s}.\ \tau \rhd j_1 \qquad \Delta \vdash i : s}{\Delta \vdash e\ i : \tau[i/a] \rhd j_1 + j + C_{\mathsf{AppT}}(n_k, b_k)}\ \forall_i\mathrm{E} \qquad \frac{\Delta \vdash \mathsf{wf}\ s \qquad e\ \text{is value} \qquad \Delta; a : s \vdash e : \tau \rhd i}{\Delta \vdash \Lambda a : s.\ e : \forall^{C_{\mathsf{AbsTInner}}(e)}_{a:s}.\ \tau \rhd C_{\mathsf{AbsT}}(i, e)}\ \forall_i\mathrm{I}$$

$$\frac{\Delta \vdash (\exists\alpha :: \kappa.\ \tau) :: * \qquad \Delta \vdash \tau_1 :: \kappa \qquad \Delta \vdash e : \tau[\tau_1/\alpha] \rhd i}{\Delta \vdash \mathsf{pack}_{\exists\alpha::\kappa.\ \tau}\ \langle \tau_1, e \rangle : \exists\alpha :: \kappa.\ \tau \rhd i + C_{\mathsf{Pack}}}\ \exists\mathrm{I}$$

$$\frac{\Delta \vdash e_1 : \exists\alpha :: \kappa.\ \tau \rhd i_1 \qquad \Delta; \alpha :: \kappa; x : \tau \vdash e_2 : \tau_2 \rhd i_2 \qquad \alpha \notin \mathsf{FTV}(\tau_2)}{\Delta \vdash \mathsf{unpack}\ e_1\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e_2 : \tau_2 \rhd i_1 + i_2 + C_{\mathsf{Unpack}}}\ \exists\mathrm{E}$$

Figure 2-10: Typing rules

$$\frac{\Delta \vdash (\exists a : s.\ \tau) :: * \qquad \Delta \vdash i :: s \qquad \Delta \vdash e : \tau[i/a] \triangleright j}{\Delta \vdash \mathsf{pack}_{\exists a:s.\ \tau}\ \langle i, e\rangle : \exists a : s.\ \tau \triangleright j + C_{\mathsf{Pack}}}\ \exists_i\mathrm{I}$$

$$\frac{\Delta \vdash e_1 : \exists a : s.\ \tau \triangleright i_1 \qquad \Delta; a : s; x : \tau \vdash e_2 : \tau_2 \triangleright i_2 \qquad a \notin \mathsf{FIV}(\tau_2, i_2)}{\Delta \vdash \mathsf{unpack}\ e_1\ \mathsf{as}\ \langle a, x\rangle\ \mathsf{in}\ e_2 : \tau_2 \triangleright i_1 + i_2 + C_{\mathsf{Unpack}}}\ \exists_i\mathrm{E}$$

$$\frac{\Delta \vdash e_m : o_{\mathsf{bt}}.\tau_m \triangleright i_m\ (m = 1, 2)}{\Delta \vdash e_1\ o_{\mathsf{bt}}\ e_2 : o_{\mathsf{bt}}.\tau_{\mathsf{res}} \triangleright i_1 + i_2 + C_{\mathsf{BinOp}}(o_{\mathsf{bt}})}\ \textsc{BinOp} \qquad \frac{\Delta \vdash e_m : \mathsf{nat}\ i_m \triangleright j_m\ (m = 1, 2)}{\Delta \vdash e_1 \oplus e_2 : \mathsf{nat}\ (i_1 + i_2) \triangleright j_1 + j_2 + C_{\mathsf{NatPlus}}}\ \textsc{Nat}_+$$

$$\frac{\Delta \vdash e_k : \tau \triangleright i_k\ (k = 1, \cdots, n) \qquad \mathsf{goodWidth}(m, \tau)}{\Delta \vdash \mathsf{arrayFromList}_m\ \{e_1, \cdots, e_n\} : \mathsf{array}_m\ \tau\ n \triangleright i_1 + \cdots + i_n + C_{\mathsf{ArrayFromList}}(m, n)}\ \textsc{New1}$$

$$\frac{\Delta \vdash e_1 : \mathsf{nat}\ i \triangleright j_1 \qquad \Delta \vdash e_2 : \tau \triangleright j_2 \qquad \mathsf{goodWidth}(n, \tau)}{\Delta \vdash \mathsf{new}_n\ e_1\ e_2 : \mathsf{array}_n\ \tau\ i \triangleright j_1 + j_2 + C_{\mathsf{New}}(n, i)}\ \textsc{New2}$$

$$\frac{\Delta \vdash e_1 : \mathsf{array}_n\ \tau\ i_1 \triangleright j_1 \qquad \Delta \vdash e_2 : \mathsf{nat}\ i_2 \triangleright j_2 \qquad \Delta \vdash i_2 < i_1}{\Delta \vdash \mathsf{read}_n\ e_1\ e_2 : \tau \triangleright j_1 + j_2 + C_{\mathsf{Read}}(n)}\ \textsc{Rd}$$

$$\frac{\Delta \vdash e_1 : \mathsf{array}_n\ \tau\ i_1 \triangleright j_1 \qquad \Delta \vdash e_2 : \mathsf{nat}\ i_2 \triangleright j_2 \qquad \Delta \vdash i_2 < i_1 \qquad \Delta \vdash e_3 : \tau \triangleright j_3}{\Delta \vdash \mathsf{write}_n\ e_1\ e_2\ e_3 : \mathsf{unit} \triangleright j_1 + j_2 + j_3 + C_{\mathsf{Write}}(n)}\ \textsc{Wr}$$

$$\frac{\Delta \vdash e : \mathsf{array}\ \tau\ i \triangleright j}{\Delta \vdash \mathsf{len}\ e : \mathsf{nat}\ i \triangleright j + C_{\mathsf{Len}}}\ \textsc{Len} \qquad \frac{\Delta(\ell) = (\tau, i)}{\Delta \vdash \ell : \mathsf{array}\ \tau\ i \triangleright 0}\ \textsc{Loc}$$

$$\frac{\Delta \vdash e : \tau \triangleright i_1 \qquad \Delta \vdash i_2 : \mathsf{Time} \qquad \Delta \vdash i_1 \le i_2}{\Delta \vdash e : \tau \triangleright i_2}\ \textsc{Relax} \qquad \frac{\Delta \vdash e : \tau_1 \triangleright i \qquad \Delta \vdash \tau_2 :: * \qquad \Delta \vdash \tau_1 \equiv \tau_2 :: *}{\Delta \vdash e : \tau_2 \triangleright i}\ \textsc{TyEq}$$

Figure 2-11: Typing rules (continued)

first argument or the result, written as $o.\tau_1$ and $o.\tau_{\mathsf{res}}$.

Typing judgments have the form $\Delta \vdash e : \tau \triangleright i$ where i represents an upper bound of time needed to reduce e to a value[8]. i is an open index that may refer to index variables in Δ. Typing rules are defined preserving an invariant that if $\Delta \vdash e : \tau \triangleright i$ then τ is of kind $*$ and i is of sort Time.

The introduction and elimination rules $\rightarrow$I and $\rightarrow$E for function types reflect the intuition of how to count beta reductions. According to rule $\rightarrow$E, the running time of a function is the running time of the function body, and the time to evaluate a function application $e_1\ e_2$ is the sum of that needed for e_1, e_2, the function body, plus some extra cost for the beta reduction. Most rules in Figure 2-10 involve some cost parameters which will be defined in Section 4.5. In the simple cost model used in this chapter, $C_{\mathsf{App}}(_,_) = 1$ and all other cost parameters are zero.

The rule Rec for fixpoints requires that the body be a function abstraction, possibly wrapped by some index polymorphism, which ensures that any two consecutive fixpoint unrollings will trigger at least one beta reduction. Note that TiML supports index-polymorphic recursion where the index argument can change in a recursive call. Being able to make a recursive call with a different index is necessary to reflect the change of argument size.

In rules μI and μE, c denotes either a type or an index, so $\overrightarrow{c}$ denotes a list of mixed types and indices. $\mathsf{FTV}(\cdot)$ and $\mathsf{FIV}(\cdot)$ stand for free type and index variables respectively. Rules Rd and Wr for array read and write perform static bound checking by requiring $\Delta \vdash i_2 < i_1$. Structural (non-syntax-directed) rules Relax and TyEq are for relaxing the time bound and using an equivalent type.

Note that the form restriction in rule Rec means that fixpoints always define functions, and terms such as recursive lists are ruled out. The constraint is also present in SML, since in SML one can only define a fixpoint by the "fun" keyword, which defines a function that must take at least one argument.

In rule Ifi, note that variable x is given the type $\exists\{j = \mathsf{true}\}.\ \mathsf{unit}$ when type-

[8]Let us ignore the memory costs for now. When memory costs are considered, i is a pair of indices, of sorts Time and Nat.

checking the first branch e_1[9]. e_1 can add the premise $j = \textsf{true}$ into the typechecking (and VC-checking) context by unpacking x. Rules New1 and New2 use a function goodWidth, defined below, to check that the element type and the element width are compatible[10].

$$
\begin{aligned}
\textsf{goodWidth}(1, \textsf{byte}) &= \textsf{true} \\
\textsf{goodWidth}(32, _) &= \textsf{true} \\
\textsf{goodWidth}(_, _) &= \textsf{false}
\end{aligned}
$$

2.3.2 Typing examples

To help the reader build the right intuition about the syntax and typing rules, I write the fold-left example in the formal syntax:

$$
\begin{aligned}
\textsf{list} &\stackrel{\textsf{def}}{=} \mu\gamma : * \Rightarrow \textsf{Nat} \Rightarrow *.\ \lambda\alpha : *.\ \lambda a : \textsf{Nat}.(\exists_ : \{_|a = 0\}.\ \textsf{unit}) + \\
&\qquad\qquad\qquad\qquad (\exists b : \textsf{Nat},\ _ : \{_|a = b + 1\}.\ \alpha \times \gamma\ \alpha\ b) \\
\textsf{foldl} &\stackrel{\textsf{def}}{=} \Lambda\alpha\ \beta : *.\ \textsf{rec}\ g.\ e \\
e &\stackrel{\textsf{def}}{=} \Lambda m\ n : \textsf{Nat}.\ \lambda f : \alpha \times \beta \xrightarrow{m} \beta.\ \lambda y : \beta.\ \lambda l : \textsf{list}\ \alpha\ n. \\
&\quad\ \textsf{case unfold}\ l\ \textsf{of} \\
&\qquad z.\textsf{unpack}\ z\ \textsf{as}\ \langle_, _\rangle\ \textsf{in}\ y \\
&\quad\ \textsf{or}\ z.\textsf{unpack}\ z\ \textsf{as}\ \langle n', w\rangle\ \textsf{in unpack}\ w\ \textsf{as}\ \langle_, u\rangle\ \textsf{in}\ g\ m\ n'\ f\ (f(u.1, y))\ u.2 \\
e &: \forall m\ n : \textsf{Nat}.\ (\alpha \times \beta \xrightarrow{m} \beta) \xrightarrow{0} \beta \xrightarrow{0} \textsf{list}\ \alpha\ n \xrightarrow{T\ m\ n} \beta \\
T &\stackrel{\textsf{def}}{=} \lambda m\ n.\ (m + 4) \times n.
\end{aligned}
$$

Comparing with the source code in Listing 2-1, we can see that datatypes are translated into recursive types where the restriction on the index argument is translated into a refinement in each constructor. The unnamed index of the refinement sort as well as any extra index arguments in each constructor are existentially quantified. Pattern matching is translated into unfolding, case-analysis, and series of unpackings, the last of which makes the refinements in each constructor available in that branch.

[9] $\exists\{j = \textsf{true}\}.\textsf{unit}$ is a simplified syntax for $\exists a : \{a : \textsf{Unit}|j = \textsf{true}\}.\textsf{unit}$, meaning the index packed in this type, a, is of sort $\{a : \textsf{Unit}|j = \textsf{true}\}$.

[10] Currently only the byte type has a width of 1; all other types (including type variables) have width 32. In other words, in order to create a 1-byte array, the element type must be known to be byte. Note that a 32-byte array of bytes is also allowed.

$\boxed{\Sigma \vdash h}$

$$\frac{\forall \ell\ \tau\ i.\ \Sigma(\ell) = (\tau, i) \to \exists \vec{v}.\ h(l) = \vec{v} \wedge |\vec{v}| = [\![i]\!] \wedge \forall v \in \vec{v}.\ (\cdot, \cdot, \cdot, \Sigma) \vdash v : \tau \triangleright 0}{\Sigma \vdash h} \text{ Heap}$$

$\boxed{\Sigma \vdash \sigma : \tau \triangleright i}$

$$\frac{(\cdot, \cdot, \cdot, \Sigma) \vdash e : \tau \triangleright i \qquad \vdash \mathsf{wf}\ \Sigma \qquad \Sigma \vdash h \qquad [\![i]\!] \le r}{\Sigma \vdash (h, e, r) : \tau \triangleright i} \text{ Config}$$

Figure 2-12: Configuration typing

The running time of each branch should be bounded by the overall bound of the function. The first branch requires us to prove the trivial VC: $0 \le T\ m\ n$; the second branch requires us to prove: $n = n' + 1 \to m + 4 + T\ m\ n' \le T\ m\ n$.

Another illustrative example is a diverging recursive function, which is not typable in TiML:

$$\mathsf{rec}\ f.\ \Lambda a : \mathsf{Nat}.\ \lambda x : \mathsf{unit}.\ f\ (a - 1)\ x$$

Its untypability is implied by the soundness theorem, which guarantees that every well-typed TiML program terminates. A more intuitive explanation is that one of the VCs it generates is

$$\forall a : \mathsf{Nat}.\ a - 1 + 1 \le a$$

which is not true without the premise $a \ge 1$ (our subtraction for Nat and Time is bounded below by zero). A proper structural recursion like foldl typechecks because the Cons branch gives us the premise $n = n' + 1$ (hence $n \ge 1$), which we do not have here (refining a to $\{a \mid a \ge 1\}$ will not work because the call $f\ (a - 1)$ will be rejected).

2.3.3 Soundness theorem

The soundness theorem of TiML states the usual "nonstuckness" property that "well-typed terms cannot get stuck." It uses configuration typing defined by rule Config in Figure 2-12, which means the term and heap are well-typed, and the available fuel is no lower than what is statically estimated by the type system.

Definition 1 (Unstuck). *A configuration σ is unstuck iff $\sigma.2$ is a value or there exists σ' such that $\sigma \rightsquigarrow \sigma'$.*

Theorem 1 (Soundness). *For all Σ, τ, i, σ, and σ', if $\Sigma \vdash \sigma : \tau \triangleright i$ and $\sigma \rightsquigarrow^* \sigma'$, then σ' is unstuck.*

Section 2.5 sketches the proof (mechanized in Coq).

Note that in rule Config, I only require that if a location is well-typed then it contains a value of the expected type. I do not need to require the other way around (i.e. if a location contains a value then it should be well-typed). The intuition is that a heap type Σ is just an underspecification of the actual heap h. Well-typed programs will only access locations in h that are specified by Σ. h can have ill-typed junk values outside Σ's domain, and they will not affect programs' behavior.

2.3.4 Decidability

I aim for relative decidability, meaning that TiML typechecking *always* produces VCs that are true iff the original program should typecheck, even though the VCs do not obviously fall in a decidable theory. The relative decidability can be witnessed by a syntax-directed algorithmic version of the typing rules in Figure 2-10 by inlining rules Relax and TyEq. In practice, I use SMT solvers to decide the VCs, which are in theory not SMT-decidable because of nonlinear formulas like $m \times n$ from e.g. time bounds or array-bound checking, though Z3 seems to be pretty good at handling them on all the benchmarks.

2.4 Typechecker implementation and big-O inference

I have implemented TiML's typechecker in SML. The typechecker is implemented from scratch, not using existing parser or typechecker implementations for similar languages, for maximal flexibility. The typechecking algorithm is based on a syntax-directed version of Figure 2-10, which is already almost syntax-directed except for rules Relax and TyEq that can be inlined into other rules.

The typechecker supports Hindley-Milner type inference and some index inference, particularly inferring big-O classes from recurrences. Types and indices can be omitted with underscores, and the typechecker generates unification variables (abbreviated as *uvars* from here on) in place of these underscores. Type uvars are unified or generalized during typechecking per the Hindley-Milner algorithm; after typechecking there should not be any type uvars left. Index uvars are unified as much as possible during typechecking, though after typechecking there could remain some index uvars in the program and in VCs. Index uvars in VCs are converted into existentially quantified variables, and these VCs with existential quantifiers are sent to our heuristic pattern-matching-based recurrence solver.

With full annotations of the running time of recursive functions, the VCs are regular inequality formulas like $3(n-1)+3 \leq 3n$. If the programmer decides that coming up with a time-complexity annotation like $3n$ is too burdensome, she can choose to omit this annotation, and the inference-enabled typechecker will generate VCs like $T(n-1)+3 \leq T(n)$ with an unknown $T(\cdot)$. In this situation we face the problem of recurrence solving, which TiML handles in an incomplete way by using heuristic pattern-matching-based big-O complexity inference that can handle recurrences resulting from many common iteration and divide-and-conquer patterns (because it is heuristics-based, I do not have a clear characterization of its applicability, though). For example, seeing a pattern $T(n-1)+3 \leq T(n)$, the solver infers that the function's time complexity is $O(n)$; seeing a pattern $2T(\lfloor n/2 \rfloor)+4n+5 \leq T(n)$, the solver infers that the function's time complexity is $O(n \log(n))$.

The task of the recurrence solver is to massage the VCs in order to find patterns such as $T(n-1)+3 \leq T(n)$ (the searching of patterns is a shallow syntactical matching and there is no notion of canonical forms)[11]. The solver first lifts all irrelevant conjuncts out of existential quantifiers, so that under each existential quantifier are only conjuncts that are relevant in finding a value for the existential variable. Then it looks for VCs of the forms

$$\exists T.\, T \leqq g \wedge \forall m,n.\, A \leq T(m,n) \quad \text{or} \quad \exists g,T.\, T \leqq g \wedge \forall m,n.\, A \leq T(m,n).$$

The first form corresponds to the case where the programmer has provided a big-O specification (e.g. `msort`), while the second case corresponds to where the programmer has omitted the big-O class (e.g. `split`). The programmer-provided g will be ignored first in the first form, and the solver will come up with its own inference of g, which will be compared to the programmer-supplied specification.

The bulk of the solver's work is analyzing the A part. It treats A as a sum of terms and tries to find among these terms those of the form $T(m, \lceil q_i/b \rceil)$ or $T(m, \lfloor q_i/b \rfloor)$ with a common divisor b but possibly different q_i's where $q_i \leq n$. It uses an SMT solver to do equality and inequality tests to be more robust. After collecting these "sub-problem" terms, it tries to find the big-O classes of the remaining terms. A big-O class has the form $mn^c \log^d n$. Some terms have their big-O classes in the premise context, such as T_{split} and T_{merge} in (2.1.0.1). Big-O classes are easy to combine for addition, multiplication, logarithm, and max. Finally, these collected terms are used to match the cases of the Master Theorem, which is shown below as a reminder.

Theorem 2 (Master Theorem)**.** *For recurrence $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$:*
(1) if $f(n) \in O(n^c)$ where $c < \log_b a$, then $T(n) \in \Theta(n^{\log_b a})$;
(2) if $f(n) \in \Theta(n^c \log^k n)$ where $c = \log_b a$ and $k \geq 0$, then $T(n) \in \Theta(n^c \log^{k+1} n)$;
(3) if $f(n) \in \Omega(n^c)$ where $c > \log_b a$, and $af(n/b) \leq kf(n)$ for some $k < 1$ and sufficiently large n, then $T(n) \in \Theta(f)$.

[11] A pattern-matching-based solver has the strength of being flexible and versatile, despite the weakness of being fragile in the face of superficial syntactical irregularities.

To apply it to inequality recurrences, I changed Θ to O in the conclusion to get a sound (but sometimes not tight) bound. The Master Theorem gives a solution for g (an asymptotic bound), but no solution for T (a concrete bound), so subsequent use of T cannot rely on its properties other than being of $O(g)$. Aside from using the Master Theorem for divide-and-conquer-like VCs, the solver also looks for VCs of the form $T(n-1)+O(f(n)) \leq T(n)$ and infers big-O class $T(n) \in O(nf(n))$. This heuristic is useful for simple "remove one" recursion schemes common in list processing. As can be seen from the above procedure, only a limited form of multivariate complexities is supported, namely these where n is the main variable and m is just a passive factor. Such passive factors are mainly used in cases where an algorithm takes in a primitive operation as parameter (like the `le` comparator taken in by function `msort`). Another limitation is that bounds such as $n \log n$ are only supported through this big-O mechanism. The programmer cannot specify a precise $n \log n$-like bound (not a big-O class) and tell the SMT solver to discharge the VCs, since I have not been able to teach the SMT solver to discharge VCs involving $n \log n$.

The pattern-matching-based recurrence solver is versatile albeit fragile. If the syntactic form of the VC is not something I anticipate, the solver tends to fail to recognize the recurrence pattern. One such failure is given in Section 5.1 on benchmark `array-kmed`.

2.5 Formal soundness proof

I formalized TiML and its soundness proof in Coq, constituting the first mechanized soundness proof for a resource-aware type system. The Coq proof can be found in file `proof/Soundness.v` in the source-code tarball associated with this thesis as supplemental material. The Soundness Theorem (Theorem 1 in Section 2.3.3, Theorem `soundness` in the Coq proof) is proved by the usual "preservation + progress" approach, with the preservation and progress lemmas shown below. $[\![i]\!]$ stands for the denotational semantics (i.e. interpretation) of index i, explained later in this section.

Lemma 1 (Progress). *For all Σ, τ, i, and σ, if $\Sigma \vdash \sigma : \tau \triangleright i$, then* $\mathsf{unstuck}(\sigma)$.

Proof. Induction on the typing derivation. See Coq proof of lemma `progress` for details. □

Lemma 2 (Preservation)**.** *For all Σ, τ, i, σ, and σ', if $\Sigma \vdash \sigma : \tau \triangleright i$ and $\sigma \rightsquigarrow \sigma'$, then there exist Σ' and i' such that $\Sigma' \vdash \sigma' : \tau \triangleright i'$.*

Proof. Appeal to Lemmas 3 and 4. □

Lemma 3 (Atomic Preservation)**.** *For all Σ, τ, i, σ, and σ', letting Δr be $\sigma.3 - \sigma'.3$, if $\Sigma \vdash \sigma : \tau \triangleright i$ and $\sigma \twoheadrightarrow \sigma'$, then $[\![i]\!] \geq \Delta r$ and there exists Σ' such that $\Sigma' \vdash \sigma' : \tau \triangleright (i - \Delta r)$ and $\Sigma \subseteq \Sigma'$.*

Proof. Induction on the atomic stepping derivation (the second premise). See Coq proof of lemma `preservation_atomic` for details. □

Lemma 4 (Ectx Typing)**.** *Let e be $E[e_1]$ (the plugging of term e_1 into evaluation context E). For all Σ, τ, and i, letting Δ be $(\cdot, \cdot, \cdot, \Sigma)$, if $\Delta \vdash e : \tau \triangleright i$ and $\vdash \mathsf{wf}\ \Delta$, then there exist τ_1 and i_1 such that $\Delta \vdash e_1 : \tau_1 \triangleright i_1$, $[\![i_1]\!] \leq [\![i]\!]$, and for all e_1', Σ', and i_1', letting Δ' be $(\cdot, \cdot, \cdot, \Sigma')$ and e' be $E[e_1']$, if $\Delta' \vdash e_1' : \tau_1' \triangleright i_1'$, $\vdash \mathsf{wf}\ \Delta'$, $[\![i_1']\!] \leq [\![i_1]\!]$, and $\Sigma \subseteq \Sigma'$, then $\Delta' \vdash e' : \tau \triangleright i - i_1 + i_1'$.*

Proof. Induction on the definition of the context-plugging operation. See Coq proof of lemma `ectx_typing` for details. □

Lemma 3 is the version of the preservation lemma for atomic steps, also strengthened with the time bound on the post-configuration explicitly specified as $i - \Delta r$. Lemma 4 is a characterization of the typing property of evaluation contexts. It says that if a compound term is well-typed, then the inner term is well-typed, and if one replaces the inner term with another term of the same type, the type of the compound term will not change. It also reflects the intuition that the running time of a compound term is the running time of the inner term plus that of the evaluation context. The proofs of the above lemmas make use of various versions (for sorting/kinding/typing) of substitution lemmas, canonical-value-form lemmas, and weakening lemmas, for each of which I show one example below. Another important

lemma is the invariant that the typing judgments guarantee the result type and time are of proper kind/sort.

Lemma 5 (Substitution)**.** *For all* Δ*,* e_1*,* τ_1*,* i_1*,* e_2*,* x*, and* τ*, if* $\Delta, x : \tau \vdash e_1 : \tau_1 \triangleright i_1$*,* $\Delta \vdash e_2 : \tau \triangleright 0$ *and* $\vdash \mathsf{wf}\ \Delta$*, then* $\Delta \vdash e_1[e_2/x] : \tau_1 \triangleright i_1$*.*

Proof. Induction on the typing derivation for e_1. See Coq proof of lemma `typing_subst_e_e` for details. □

Note that in the above lemma I require time 0 in e_2. The reason is that variable x may have multiple appearances in e_1. If e_2 has nonzero running time, then after the substitution there may be multiple copies of e_2, and the resulting time will not be simply $i_1 + i_2$. I got away with fixing i_2 to 0 because in all the proofs, term substitution only happens when the substitute is a value.

Lemma 6 (Canonical Value Form)**.** *For all* Σ*,* v*,* τ_1*,* τ_2*,* i*, and* i'*, letting* Δ *be* $(\cdot, \cdot, \cdot, \Sigma)$*, if* $\Delta \vdash v : \tau_1 \xrightarrow{i} \tau_2 \triangleright i'$ *and* $\vdash \mathsf{wf}\ \Delta$*, then there exists* e *such that* $v = \lambda x.\ e$*.*

Proof. Induction on the typing derivation. See Coq proof of lemma `canon_TArrow` for details. □

Lemma 7 (Weakening)**.** *For all* Δ*,* e*,* τ*,* i*,* x*, and* τ'*, if* $\Delta \vdash e : \tau \triangleright i$*, then* $\Delta, x : \tau' \vdash e : \tau \triangleright i$*.*

Proof. Induction on the typing derivation. See Coq proof of lemma `typing_shift_e_e` for details. □

Lemma 8 (Typing-Kinding)**.** *For all* Δ*,* e*,* τ*, and* i*, if* $\Delta \vdash e : \tau \triangleright i$ *and* $\vdash \mathsf{wf}\ \Delta$*, then* $\Delta \vdash \tau :: *$ *and* $\Delta \vdash i : \mathsf{Time}$*.*

Proof. Induction on the typing derivation. See Coq proof of lemma `typing_kinding` for details. □

One complication during the proof is the treatment of type equivalence. The definition of type equivalence should admit both equivalence rules (particularly transitivity) and good inversion lemmas such as Lemma 9. I follow the method in Chapter

30 of [68], defining type equivalence with congruence, reduction, and equivalence rules (see the Coq definition of `tyeq`), and then I prove inversion lemmas via a "parallel reduction" version of type equivalence that enjoys a "diamond" property. My solution is more involved because even for comparing normalized types I cannot use a syntactic equality test, for I allow two semantically equivalent indices to be treated as equal. In the Coq code I use the relation `cong` to compare normalized types, which uses a semantic-equivalence test to compares indices. Because all properties about the denotational semantics (see below) of indices require well-sortedness, I need to put kinding constraints in type equivalence rules. Particularly, the transitivity rule needs a kinding constraint on the intermediate type. All judgments involving types should be morphisms on `tyeq` equivalence, expressed as lemmas such as Lemma 10.

Lemma 9 (Invert TyEq TArrow)**.** *For all Δ, τ_1, i, τ_2, τ_1', i', τ_2', and κ, if $\Delta \vdash \tau_1 \xrightarrow{i} \tau_2 \equiv \tau_1' \xrightarrow{i'} \tau_2' :: \kappa$, $\Delta \vdash \tau_1 \xrightarrow{i} \tau_2 :: \kappa$, $\Delta \vdash \tau_1' \xrightarrow{i'} \tau_2' :: \kappa$, and $\vdash \mathsf{wf}\ \Delta$, then $\Delta \vdash \tau_1 \equiv \tau_1' :: \kappa$, $\Delta \vdash i = i'$, and $\Delta \vdash \tau_2 \equiv \tau_2' :: \kappa$.*

Proof. See Coq proof of lemma `invert_tyeq_TArrow` for details. ☐

The judgment $\Delta \vdash i = i'$ used above is just an instance of the validity judgment $\Omega \vdash \theta$. Notice the dissertation's convention that I may write Δ when only its Ω part is needed.

Lemma 10 (TCtx TyEq)**.** *For all Ω, Λ, Σ, Γ, Γ', e, τ, and i, if $(\Omega, \Lambda, \Sigma, \Gamma) \vdash e : \tau \triangleright i$, $(\Omega, \Lambda) \vdash \mathsf{wf}\ \Gamma'$, Γ and Γ' have the same domain and have equivalent types at each variable, then $(\Omega, \Lambda, \Sigma, \Gamma') \vdash e : \tau \triangleright i$.*

Proof. See Coq proof of lemma `tctx_tyeq` for details. ☐

I had two unsuccessful attempts to formalize properties of type equivalence before I embarked on the current approach. The intuition of the two failed approaches was the same as the current one: bake reduction and equivalence rules into the definition and prove that the definition coincides with another relation that admits good inversion lemmas. I first tried to prove that type equivalence coincides with a set of logical relations. Logical relations are a technique for proving contextual equivalence of two

open terms, by first defining logical equivalence on closed terms and then extending it to open terms via equivalent substitutions. I ran into trouble with this approach because my types are indexed with open indices, and the index/sort system is a dependent type system. In my second attempt, I tried a fully denotational approach by interpreting open types as functions that return closed type normal forms, proving that type equivalence coincides with equivalence of those functions (assuming functional extensionality). This approach disallows impredicative polymorphic types, whose presence makes type normal forms undefinable, because the cardinality of one of the normal form's constructors (the polymorphic type case) is larger than that of the normal form itself.

The final type-equivalence definition includes three categories of rules: (1) congruence rules, for example, $\tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2'$ if $\tau_1 \equiv \tau_1'$ and $\tau_2 \equiv \tau_2'$; (2) reduction rules, for example, $(\lambda\alpha.\ \tau_1)\ \tau_2 \equiv \tau_1[\tau_2/\alpha]$; (3) equivalence rules, i.e., reflexivity, transitivity, and symmetry. For a compound example, $(\lambda\alpha.\ \tau_1 \times \alpha)\ \tau_2 \times \tau_3 \equiv \tau_1 \times \tau_2 \times \tau_3$.

In the Coq formalization, I use de Bruijn indices throughout, and since I have the three-tier index, type, and term sublanguages, I have to define multiple forms of substitution and shifting corresponding to each pair of sublanguages. A major part of the proof is establishing harmonious interactions between various forms of judgments and various forms of substitution and shifting.

Another technical hurdle is formalizing the denotational semantics of indices. The denotational semantics (i.e. the interpretation function) should have good reduction behavior to be used as an evaluator, and it needs to admit substitution lemmas such as Lemmas 11 and 12. In my Coq formalization, the interpretation function $[\![\cdot]\!]$ takes in a base-sorting context, a result base sort, and an index, and interprets the index according to the context and the result sort. For example: $[\![a+b+1]\!]^{\mathsf{Nat}}_{[a:\mathsf{Nat},b:\mathsf{Nat}]}$ = `(fun a b : nat => a+b+1)`, where `(fun a b : nat => a+b+1)` is a Coq term. The intuition is straightforward: closed indices, when interpreted in an empty context, denote just numbers (e.g. $[\![3]\!]^{\mathsf{Nat}}_{[]}$=3); open indices with variables, when interpreted in a proper context, denote functions (e.g. the $a+b+1$ example) that can also be seen as numbers that are parameterized on the values of the free variables. The result base

sort is needed to make the Coq type of $[\![\cdot]\!]$ simpler; otherwise $[\![\cdot]\!]$'s type is complexly dependent on the index. I omit the subscript when the context is empty and the superscript when it can be inferred. The denotational semantics is fixed once and for all. Please see the Coq definition of function `interp_idx` for details.

Lemma 11 (Interp Subst Index)**.** *For all Δ, a, i_1, s_1, i_2, and s_2, if $\Delta, a :: s_2 \vdash i_1 : s_1$ and $\Delta \vdash i_2 : s_2$, then $[\![i_1[i_2/a]]\!] = [\![i_1]\!]([\![i_2]\!])$.*

Proof. See Coq proof of lemma `interp_subst_i_i` for details. □

Lemma 12 (Interp Subst Prop)**.** *For all Δ, a, θ, i, and s, if $\Delta, a :: s \vdash \theta$, $\Delta \vdash i : s$, $\Delta \vdash \mathsf{wf}\,\theta$ and $\vdash \mathsf{wf}\,\Delta$, then $\Delta \vdash \theta[i/a]$.*

Proof. See Coq proof of lemma `interp_subst_i_p` for details. □

The validity judgment $\Omega \vdash \theta$ is defined as first collecting refinements in Ω as θ' and then interpreting the syntactic proposition $\theta' \to \theta$ in the meta-logic. The latter is done in a similar way as the interpretation of indices. For example, $a : \mathsf{Nat}, b : \{\mathsf{Nat} | a = b\} \vdash a + 1 = b + 1$ is defined as $\forall a\ b : \mathsf{nat},\ a = b \to a + 1 = b + 1$.

2.6 ETiML: a TiML variant for smart contracts

This chapter so far has described TiML as a general functional language with a simple cost model where each function application costs one unit of time and none of the other operations cost either time or memory. In this section I will describe ETiML (Ethereum TiML), a variant of TiML particularly designed to write Ethereum smart contracts. It has a slightly different surface syntax and a cost model that faithfully reflects the actual gas cost of smart contracts. It also adds one feature that is important in the smart-contract setting: states.

An example smart contract written in ETiML is shown in Figure 2-13, which implements a token. Each contract provides a set of public functions that can be invoked from outside. The public functions provided by contract `Token` are `transfer` for transferring tokens between accounts (each account is identified by its address)

```
contract Token = struct

  state balanceOf : map address uint256
  state outflows : vector address

  fun constructor (initialSupply : uint256) =
    set balanceOf[msg.sender] initialSupply

  public fun transfer {n : ℕ} (_to : address, _value : uint256)
      pre {outflows : n} post {outflows : n+1} =
    require(balanceOf[msg.sender] ≥ _value);
    require(balanceOf[_to] + _value ≥ balanceOf[_to]); (* Check for overflows *)
    modify balanceOf[msg.sender] -= _value;
    modify balanceOf[_to] += _value;
    push_back (outflows, msg.sender)

  public fun numOutflows {n : ℕ} () pre {outflows : n}
      (* using $(4438*n+12384), 1984*n+5568 *) =
    for (#0, vector_len outflows, 0, λ {i | 0 ≤ i ∧ i < n} (i : nat {i}, acc) ⇒
      if outflows[i] = msg.sender then acc+1 else acc end
    )

end
```

Figure 2-13: ETiML example: token

and `numOutflows` for getting the number of outgoing transfers for the calling account. Contracts use EVM's storage to store data that persist across multiple transactions (each invocation of a public function from the outside is a transaction). In ETiML, data in storage are declared using the keyword `state`. Each identifier declared by a `state` keyword is called a "state name." Contract `Token` declares two state names: `balanceOf` for recording the token balance of each account and `outflows` for recording the source address of each transfer. Each state name is associated with a type, two of which are used in this contract: `map` and `vector`. A `map` maps word-sized (512-bit) keys to values (see Section 3.4.7 for allowed map-value types). A `vector` is an in-storage array of word-sized elements.

Unlike previous TiML examples where a function's cost only depends on its arguments, in a smart contract, a function's cost can depend on the value or size of some in-storage data. For example, the cost of function `numOutflows` depends on the length of vector `outflows`, since the former does an iteration over the latter. To allow costs to depend on the lengths of vectors, I introduced the keywords `pre` and `post` in function signatures to specify the lengths of vectors before and after the function. A length is an index that can contain index variables; in this way a function can work with vectors of arbitrary lengths. For example, the signature of function `transfer` says that the function expects the length of the vector `outflows` to be `n` at its entrance, while `n` is a universally quantified natural number. When the function finishes, the length of `outflows` is guaranteed to be `n+1`.

The content in the `pre` or `post` clause is called a "state specification" ("state" for short). It is a map from state names to indices. It does not need to cover all vectors, only the ones that will be accessed by the function. Accessing a vector not present in the function's `pre` clause is a type error. When the `post` clause is omitted, it is assumed to be the same as the `pre` clause. When both are omitted, they are assumed to be empty. The formal syntax of arrow types and lambda abstractions in Figure 2-4, when written in their full form that includes pre/postconditions, are

$$\langle \phi_1, \tau_1 \rangle \xrightarrow{i} \langle \phi_2, \tau_2 \rangle \text{ and } \lambda^{\phi} x : \tau.\ e.$$

ϕ stands for an index of sort `State`, constructed via $\{\overrightarrow{u : i}\}$ or $\phi_1 \cup \phi_2$. A lambda abstraction is only annotated with a precondition because the postcondition can usually be derived.

The time cost of `numOutflows` is `4438*n+12384`, where `n` is the length of vector `outflows`. There is a second cost metric in ETiML: the amount of heap allocations (in units of bytes). "Heap" is my own derived concept implemented on top of the "memory" concept built into EVM[12]. All objects that are larger than words (e.g. tuples, arrays, etc.) are allocated on the heap. More heap allocations will lead to larger memory footprint and finally to higher gas consumption[13]. `numOutflows`'s heap-allocation amount is `1984*n+5568`.

Function `numOutflows`'s cost specifications can be inferred automatically because it is implemented with a combinator `for` instead of as a recursive function[14]. `for` here iterates from zero (inclusive) to the length of `outflows` (exclusive), calling an anonymous function (the loop body) with the loop variable `i` to calculate a result (initially `0`). `#0` is the notation for indexed natural number 0. Vector access is boundary-checked statically, so the information `0 ≤ i ∧ i < n` is necessary here.

`constructor` is a special function that is automatically called when a contract is deployed on the blockchain. `Token`'s constructor initializes the `balanceOf` map by giving all the `initialSupply` tokens to the contract creator[15]. Some minor language features are worth mentioning here. `msg.sender` is an environment variable giving the caller of the current public-function invocation[16]. `require` is a function that at runtime checks that its Boolean argument is true, throwing an exception otherwise. `modify` is syntax sugar for modifying the value of a map or vector slot by applying a unary function to it. `-= _value` is just a partial application of the binary function

[12]It is implemented using a free pointer and is never garbage-collected.

[13]See Section 3.2 for an explanation of why I choose to have time and heap allocations as two separate cost metrics instead of a single "gas" metric.

[14]`for` itself is a recursive function. `for` is a recursor (or eliminator) for indexed natural numbers, akin to `foldl` for lists.

[15]All state slots are initialized to be zero.

[16]More precisely, it is the caller responsible for this contract's current execution (or equivalently the sender of the current message). It is different from `msg.origin`, the original initiator of the current transaction. See Chapter 3 and the EVM specification for details.

`-=`[17]. `push_back` is the builtin operation to append an element at the end of a vector.

[17] `a-=b` is defined as `b-a`. The notations here are chosen to mimic C's compute-and-assign operators.

Chapter 3

TiEVM

This chapter describes TiEVM (Timed EVM), a typed assembly language with resource bounds for writing Ethereum smart contracts. TiEVM serves as the target language to compile from the TiML language described in Chapter 2. TiEVM is a typed version of the Ethereum Virtual Machine (EVM) language used by the Ethereum platform to express arbitrary computations and changes of the Ethereum world state. I will start this chapter with a brief introduction of EVM and then dive into the details of TiEVM's design choices and technical definitions.

3.1 An EVM primer

EVM is a stack-based (similar to JVM) bytecode language used by Ethereum to represent transactions (changes of world state), stack-based in the sense that all instructions operate on operands on top of the stack. EVM is Turing complete. The world state of Ethereum is a mapping from an account address (an 160-bit integer) and a word (a 256-bit integer) key to a word value. Equivalently, the world state is the collection of all accounts' storages, where each "storage" is a map from words to words (i.e. a word-addressable word array). Storage is persistent ("nonvolatile") across transactions. To make programming easier, EVM also has ephemeral ("volatile") data-storage facilities: stack and memory. The stack is a word array with a maximal length of 1024, which can only be accessed at the top 17 positions. The memory is

a map from words to bytes (i.e. a word-addressable byte array). Each transaction starts with an empty stack and an empty memory; after the transaction finishes, its stack and memory are discarded[1]. Only changes to the storage will live on.

Each account (identified by its 160-bit account address, which is usually the public key of the account owner) is associated with a piece of EVM bytecode (a "smart contract") and an ether balance. A transaction is initiated by an end user (usually a person) by sending a message with associated data to an account, which upon receiving the message runs its associated EVM code. During execution, the EVM code can send messages to other accounts (in a synchronous/blocking manner), triggering smart contracts associated with these accounts. The callee contract will run in a new stack and memory, not sharing the caller's. All those cascading executions of smart contracts are considered to be in the same transaction. A transaction ends when the computation finishes successfully, or when there is an exception thrown during the executions of the contracts, which will cause all storage changes during this transaction to be rolled back. Each message-passing can also transfer ethers from the sending account to the receiving account.[2]

As described in Section 1.3, to regulate resource (CPU and memory) usage in transaction processing, EVM has a gas mechanism. Gas consumption comes from three sources: (1) evaluating instructions, which costs CPU time; (2) touching higher memory addresses, which requires the computing node to have more memory; and (3) setting a storage cell from zero to a nonzero value, which increases the disk space needed to store the storage, because storage is sparsely encoded, and adding a nonzero value will increase the size of the encoding. To further incentivize storage saving, EVM gives gas rewards when one resets a storage cell from nonzero to zero or deletes (setting all bits to zero) an account's associated contract.

[1] More precisely, stack/memory set-up and tear-down happen at the start and end of each contract; see next paragraph.

[2] An account's ether balance can be seen as a special cell in its storage.

3.2 Design of TiEVM

The design goal of TiEVM is to have a type system for EVM that rules out traditional type-safety errors (dereferencing invalid pointers, jumping to invalid addresses, out-of-bounds array accesses, operations on wrong data types, etc.) and generates accurate and sound estimations of gas upper bounds. According to the sources of gas consumption described above, to get accurate bounds, we need to estimate (1) instruction-evaluation costs, (2) the highest memory address that will be accessed ("memory high-water mark"), and (3) the zeroness of every storage cell at each moment. The design of TiEVM centers around the first two aspects; we will discuss the third aspect later in this chapter.

TiEVM estimates instruction costs and memory high-water mark by having bounds for time and heap allocations in types ("time" here means gas cost for evaluating instructions). The total amount of heap allocations corresponds to the memory high-water mark because (1) heap-allocated objects are never freed or garbage-collected, hence new objects are allocated at ever-higher addresses; (2) non-heap memory usage is concentrated at the lower part of memory whose size will be determined at compile time. We use this lower part of memory to implement a scratch space and a register file. The maximal number of registers needed by a contract can be determined at compile time because I do whole-program (or "whole-contract") compilation.

As a result, a resource bound in TiEVM always appears as a pair of numbers, one for time and one for heap allocation. The reason why I do not combine them into a single metric is that the gas cost of memory usage is quadratic in the memory high-water mark, according to the EVM specification. Estimating the two metrics separately and only combining them at the very end (to form a total gas estimation) makes all the accounting easier.

I only use the stack for providing operands to instructions and for storing intermediate results when evaluating compound expressions; particularly, I do not use it to implement local variables or function calls. The reason is that the stack's length is limited to 1024 by the Ethereum protocol, so using it for function calls will limit

the call depth to 1024[3].

TiEVM contains some pseudo-instructions (or "macros") not present in the official EVM specification [5]. They will be expanded into pre-defined sequences of EVM instructions, listed in Appendix A.2. These pseudo-instructions allow us to define the semantics of TiEVM at a slightly higher abstraction level than EVM, talking about notions such as registers, heap allocations, branching on sum types, etc. The expansion rules are straightforward, short, and easy to scrutinize, so we feel this compromise of fidelity in order to simplify semantics is justified. If one so desires, these pseudo-instructions can be done away with by tagging the expanded instructions with markers to guide the typechecker to treat specific sequences of instructions as wholes, or to have a more involved semantics with some degree of pointer-aliasing analysis.

Which pseudo-instructions should be added is a subjective judgment. Pseudo-instructions at a higher abstraction level provide more user-friendly operations such as initializing a tuple or copying a tuple between memory and storage, but the expansion of such instructions is longer; pseudo-instructions at a lower level have shorter and simpler expansions. We draw the line where we do not support multiword instructions (such as whole-tuple or whole-array copying), so that the costs of pseudo-instructions are constant and do not depend on the types of the operands.

Instructions for calling external contracts and for emitting logging information are currently not covered by TiEVM. These instructions, when appearing in a TiEVM program, are assumed to have zero costs.

3.3 Syntax

The grammar of TiEVM is shown in Figure 3-1. TiEVM reuses indices, sorts, and the sorting rules from TiML. It reuses some of TiML's types, extending with new ones. The TiEVM-specific types are mostly level-low versions of TiML counterparts, containing some technical details that only show up at the assembly level.

[3]which means non-tail-recursive functions cannot operate on datatypes larger than 1024.

Program
$P \;::=\;$ $\mathsf{prog}\ (\{l_1 \mapsto \mathsf{block}\ \{\tau_1 : J_1\}, l_2 \mapsto \mathsf{block}\ \{\tau_2 : J_2\}, \cdots\}, J)$
Instruction Sequence
$J \;::=\;$ $I; J \mid \textsc{Jump} \mid \textsc{Halt}$
Instruction
$I \;::=\;$ $\textsc{Push}_n\ w \mid \textsc{Add} \mid \textsc{JumpI} \mid \cdots \mid \textsc{BrSum} \mid \textsc{TupleMalloc} \mid \textsc{MapPtr} \mid \cdots$
$\quad\mid\;$ $\textsc{AscTime} \mid \textsc{AppT} \mid \cdots$
Word Value
$w \;::=\;$ $() \mid d \mid \overline{n} \mid b \mid \overline{b} \mid c \mid l \mid \mathsf{state}\ n \mid \mathsf{never}_\tau$
Type
$\tau \;::=\;$ <same as TiML> $\mid (\phi, R, S) \xrightarrow{i_1, i_2} \blacksquare \mid \mathsf{tuplePtr}\ \overrightarrow{\tau}\ n\ b \mid \mathsf{arrayPtr}_n\ \tau\ j\ i \mid \mathsf{vectorPtr}\ u\ i$
$\quad\mid\;$ $\mathsf{preTuple}\ \overrightarrow{\tau}\ n\ m \mid \mathsf{preArray}_n\ \tau\ j\ i\ b_1\ b_2$

Figure 3-1: TiEVM syntax

TiEVM code is organized as basic blocks. A TiEVM program (or a TiEVM "contract") consists of a standalone instruction sequence (the "main" code) and a map from code labels to basic blocks. Each basic block consists of a block signature followed by an instruction sequence. A block signature is a type describing the precondition of the block entry point (it will always be a code-pointer type, as described in Section 3.4.3). The main code is the entry point of the program, whose precondition is always an empty environment. A instruction sequence always ends with an unconditional jump or a halting instruction. Instructions include real instructions that correspond directly to EVM instructions (e.g. Push) and pseudo-instructions (e.g. BrSum). The type system does not distinguish them. A special type of pseudo-instructions are called "noop instructions" (e.g. AscTime). Their expansions and costs are empty and zero, and their existence is only for providing some information to the typechecker. Some real instructions are also annotated with extra information, similar to annotations on expressions in a syntax-directed type system. The whole list of TiEVM instructions appears in Appendix A.1.

The Push instruction, for pushing a value onto the stack, has two parameters. w is the value to be pushed, and n is the width of the value (in number of bytes). n does not affect typechecking (it only affects the assembling of this instruction) and will mostly be omitted in this thesis. w belongs to the syntax class of "word values,"

values that can be stored in single machine words (256-bit). They include unit value $()$, integers d, natural numbers $\overline{n}$, Booleans b, indexed Booleans $\overline{b}$, bytes c, code labels l, and state names $\mathsf{state}\ n$. Booleans b and indexed Booleans $\overline{b}$ both consist of true and false; the difference is that Booleans are of type bool and indexed Booleans $\overline{b}$ are of type $\mathsf{ibool}\ b$ ($\overline{\mathsf{true}}$ is of type $\mathsf{ibool\ true}$ and $\overline{\mathsf{false}}$ is of type $\mathsf{ibool\ false}$). Similarly integers d are of type int while a natural number $\overline{n}$ is of type $\mathsf{nat}\ n$, the type of indexed natural numbers.[4]

Labels l are only used for code labels.[5] A state name $\mathsf{state}\ n$ is encoded as an integer, which in the current implementation is the place-in-order of its declaration among other state names in the source code. never_τ is a word that can be any value; its typing rule (see Section 3.4.4) makes sure that it is never used (the execution never reaches it), so it can be of any type, akin to the return type of exception throwing.

As an aside, I want to mention here that the values of existential types, recursive types, and index-/type-argument instantiations are also word-sized, because these types do not change the runtime representation of the value[6]. TiEVM has a type-erasing semantics, in the sense that type information does not have runtime consequences, so the behavior of a TiEVM program is the same as the version with all types stripped. In other words, types are only for helping the typechecker establish a proof of the well-behavedness of programs; they do not affect runtime behavior.[7]

The first TiEVM-specific type is the code-pointer type $(i, R, S) \xrightarrow{(j_1,j_2)} \blacksquare$. As the TiEVM version of function types (arrow types) in TiML, it is the type of a code pointer that specifies the precondition of the target code. It has four components. The first component, i, is an index representing the precondition on the state. The second component, R, is a register typing context, which is a map from register

[4]The n in type $\mathsf{nat}\ n$ can be any natural number, but the runtime currently can only handle word-sized natural numbers. This discrepancy will be fixed in the future by using infinite-precision integers to implement natural numbers.

[5]Heap labels do not show up in program syntax and do not concern the typechecker.

[6]Values of datatypes that have only one constructor may also be word-sized. Values of datatypes that have more than one constructors will be larger than words, because those datatypes are implemented with sum types.

[7]An opposite semantics approach would be "intensional type analysis" [44], where types have runtime data presenting them, and the execution can branch on such data.

numbers to types. It specifies the type of each register upon entering the code, akin to argument types for functions. Registers not present in R are considered to have junk values at the moment and should not be used without being initialized first. The third component, S, is a stack typing context, assigning types to stack cells. It is a list of types corresponding to the top portion of the stack. Stack cells deeper than that should not be used without initialization. The last part is a pair of indices representing the time and heap-allocation bounds for the code. As described in Section 3.2, resource bounds in TiEVM always appear as pairs, one for time and one for heap allocations. The bounds here specify the resource usage from entering the code to the very end of the program's execution; how much of that is used by this code will be determined by the type of the return pointer (a code pointer that this code jumps to when it finishes). In TiEVM, resource bounds always talk about resource usage from this point to "the very end," following a continuation-passing style.

The other TiEVM-specific types are all pointers to data containers (tuples, arrays, and vectors). In TiEVM, accessing data in a data container is done by (1) obtaining a pointer to the container, (2) doing pointer arithmetic, and then (3) reading or writing at the position pointed to by the pointer. As a result, every pointer type is in the form of an "offset" plus some information about the whole container. The offset is manipulated by pointer arithmetic; reading/writing will be regulated by container-wise information (e.g. a bounds check). $\mathsf{tuplePtr}\ \overrightarrow{\tau}\ n\ b$ is the type of pointers pointing into tuples. $\overrightarrow{\tau}$ gives the types of the tuple's components. n is the offset. b is a Boolean flag telling whether this tuple is on storage. On-storage and on-memory tuples have different sets of operations and are considered different types. $\mathsf{arrayPtr}_n\ j\ i$ is the type of pointers into arrays. n is the width of each array element, in bytes. The first index j is the length of the array while the second index i is the offset. Unlike tuple pointers where an offset n is a concrete natural number, an array pointer's offset is an index whose value may be symbolic and unknown at compile time. $\mathsf{vectorPtr}\ u\ i$ is the type of vector pointers, consisting only of the vector name u and the offset i. Vectors and other storage facilities will be described in detail in Section 3.4.7.

Sorting Context	Kinding Context	Register Context
$\Omega \;:\; \{\overrightarrow{a \mapsto s}\}$	$\Lambda \;:\; \{\overrightarrow{\alpha \mapsto \kappa}\}$	$R \;:\; \{\overrightarrow{n \mapsto \tau}\}$
Stack Type	State Spec.	Full Context
$S \;:\; \overrightarrow{\tau}$	$\phi \;=\; i$	$\Delta \;=\; (\Omega, \Lambda, R, S, \phi)$
Code Labels	State-Name Decipher	State Types
$H \;:\; \{\overrightarrow{l \mapsto \tau}\}$	$L \;:\; \{\overrightarrow{n \mapsto u}\}$	$T \;:\; \{\overrightarrow{u \mapsto \tau}\}$
Global Information		
$G \;=\; (H, L, T)$		

Figure 3-2: TiEVM typing contexts

The types $\mathsf{preTuple}\ \overrightarrow{\tau}\ n\ m$ and $\mathsf{preArray}_n\ \tau\ j\ i\ b_1\ b_2$ are used for initializing tuples and arrays, which will be explained when I describe their typing rules in Section 3.4.6.

3.4 Typing rules

Selected typing rules of TiEVM are listed in several figures from Figure 3-3 to 3-9. The rules use notations and conventions described in Section 3.4.1. I organized the rules into several groups, each centering on one aspect of the language or one type of operations. The typing system of TiEVM is relatively complex, with many kinds of rules and typing contexts needed to describe all the details of a low-level system[8]. I tried to hide as many irrelevant details as possible to flatten the learning curve. It should be noted that although the type system is complicated by the low-level details, all of them will be derived by the compiler from the source program fully automatically.

3.4.1 Notations and conventions

There are many forms of rules. The central one is the rules for typing individual instructions, with the form $G|\Delta \vdash I : \Delta' \triangleright (i_1, i_2)$, defined in Figure 3-3. G is the global information that will not change during typechecking, while Δ stands for typing contexts that could be changed after typechecking each instruction (Δ' is the result contexts). This is an important difference from type systems for high-level languages,

[8] This is a common trait of real-world low-level systems.

where typechecking expressions does not usually change typing contexts. (i_1, i_2) is the resource-bound pair corresponding to this instruction. In this thesis, I often use a single letter such as i to refer to the pair, and I write calculations such as $i + j$ and $\max(i, j)$ to mean the addition and max of pairs, meaning performing the calculation on each component. From context it should be clear whether I mean a single index or a pair of indices. I will also write 0 for $(0, 0)$.

The changeable typing context Δ consists of five components, Ω, Λ, R, S, and ϕ. They are index context, which gives the sort of each index variable; type context, which gives the kind of each type variable; register typing context, which gives the type of each register; stack typing context, which gives the type of each stack cell; and state specification ("state" for short), which is an index[9]. Because there are so many contexts, I will omit contexts that are not used and not changed in a rule. For example, rule AddTuple omits all contexts except for the stack context S. I will use $\cdot$ when the context is required to be empty. The definitions of these contexts are collected in Figure 3-2. I use commas to separate difference contexts and semicolon to mean the addition of a new element into the context (variable names added to a context are always assumed to be fresh; internally they are encoded using de Bruijn indices that build in freshness). I use $m[k \mapsto v]$ to mean updating a map m with a new key-value pair (k, v), where k can be a new key not present in m.

The unchangeable global context G consist of three components, H, L, and T. They are code-label typing context (a.k.a. a "code-heap" typing context), mapping code labels to types representing the signatures of the code; state name decipher, which maps integers to state names (needed because state names in TiEVM code are encoded as integers); and state typing context, mapping state names to types. I use a bar to separate unchangeable contexts from changeable contexts. As mentioned before, unused contexts will be omitted, and when none of the unchangeable contexts are used, the bar is also omitted (such as in rule AddTuple).

The instruction typing rules rely on many other forms of rules, such as the word-

[9]Remember from Figure 2-4 that an index can take the form of $\{\overrightarrow{u \mapsto i}\}$, a map mapping state names to indices representing the sizes or values of the corresponding state names.

value typing rules $L, T|\Omega, \Lambda \vdash w : \tau$, the type-equivalence rules $\Omega, \Lambda \vdash \tau_1 \equiv \tau_2$ and the proposition-validity rules $\Omega \vdash p$. The word-value typing rules are discussed in Section 3.4.4. Proposition-validity rules are the same as in TiML. Type-equivalence rules also overlap significantly with TiML's type-equivalence rules. There is a register-subtyping judgment $\Omega, \Lambda \vdash R_1 <: R_2$, which holds when the domain of R_1 includes the domain of R_2 and their types on each shared register are equivalent. The stack-equivalence judgment $\Omega, \Lambda \vdash S_1 \equiv S_2$ just means that the two lists have the same length and the types at each position are equivalent.

Typing rules for instruction sequences are in the form $G|\Delta \vdash J \triangleright (i_1, i_2)$. The resource bounds here mean the total consumption till the very end of the program, as in code-pointer types. There is no result context, since the postcondition of this sequence is specified by the precondition of its jump target (the return pointer).

3.4.2 Sequences and jumps

Typing rules for sequences and jumps are listed in Figure 3-3. Rule Cons is the base rule for typing instruction sequences, using typing rules for individual instructions. It is mostly self-explanatory. This is the default rule for instruction sequences, which only applies when no other rules (such as rule BrSum) apply. Rule Halt is for halting the current contract, returning the value on top of the stack. The entire stack at this point is required to contain only one value, of the type τ specified by the Halt instruction.[10]

Rule Jump is for unconditional jump. The jump target is at the top of the stack, whose type is τ (as indicated by the stack context $\tau; S$), which must be a code-pointer type $(\phi', R', S') \xrightarrow{i} \blacksquare$. In order to make a valid jump, the typechecker checks that the current register context is a subtype of the target's register-context specification, meaning that the current context is equivalent to the spec but can contain more registers; that the current stack context is equivalent to the target's specification;

[10] Halt is a pseudo-instruction which is almost equivalent to the real Return instruction. The difference is that Return accepts a piece of data on memory as the return value. Halt is implemented using Return.

$$\boxed{H, L, T|\Omega, \Lambda, R, S, \phi \vdash I : \Omega', \Lambda', R', S', \phi' \triangleright (i_1, i_2)} \quad \text{and} \quad \boxed{H, L, T|\Omega, \Lambda, R, S, \phi \vdash J \triangleright (i_1, i_2)}$$

$$\frac{\Delta \vdash I : \Delta' \triangleright i_1 \qquad \Delta' \vdash J \triangleright i_2}{\Delta \vdash I; J \triangleright i_1 + i_2} \text{ Cons} \qquad \frac{\Omega, \Lambda \vdash \tau :: * \qquad \Omega, \Lambda \vdash S \equiv [\tau]}{\Omega, \Lambda, S \vdash \text{Halt}_\tau \triangleright C_{\text{Halt}}} \text{ Halt}$$

$$\frac{\tau = (\phi', R', S') \xrightarrow{i} \blacksquare \qquad \Omega, \Lambda \vdash R <: R' \qquad \Omega, \Lambda \vdash S \equiv S' \qquad \Omega \vdash \phi \equiv \phi'}{\Omega, \Lambda, R, \tau; S, \phi \vdash \text{Jump} \triangleright C_{\text{Jump}} + i} \text{ Jump}$$

$$\frac{\begin{array}{c}\tau = (\phi', R', S') \xrightarrow{i_2} \blacksquare \\ \Omega, \Lambda \vdash R <: R' \qquad \Omega, \Lambda \vdash S \equiv S' \qquad \Omega \vdash \phi \equiv \phi' \qquad \Omega, \Lambda, R, S, \phi \vdash J \triangleright i_1\end{array}}{\Omega, \Lambda, R, \tau; \mathsf{bool}; S, \phi \vdash \text{JumpI}; J \triangleright C_{\text{JumpI}} + \max(i_1, i_2)} \text{ JumpI}$$

$$\frac{\begin{array}{c}\tau = (\phi', R', S') \xrightarrow{i_2} \blacksquare \qquad \Omega, \Lambda \vdash R <: R' \\ \Omega, \Lambda \vdash (\exists\{i = \mathsf{true}\}.\mathsf{unit}); S \equiv S' \qquad \Omega \vdash \phi \equiv \phi' \qquad \Omega, \Lambda, R, (\exists\{i = \mathsf{false}\}.\mathsf{unit}); S, \phi \vdash J \triangleright i_1\end{array}}{\Omega, \Lambda, R, \tau; \mathsf{ibool}\ i; \mathsf{unit}; S, \phi \vdash \text{JumpI}; J \triangleright C_{\text{JumpI}} + \max(i_1, i_2)} \text{ JumpIEx}$$

$$\frac{\begin{array}{c}\tau = (\phi', R', S') \xrightarrow{i_2} \blacksquare \qquad \Omega, \Lambda \vdash R <: R' \\ \Omega, \Lambda \vdash (\mathsf{ibool\ true} \times \tau_2); S \equiv S' \qquad \Omega \vdash \phi \equiv \phi' \qquad \Omega, \Lambda, R, (\mathsf{ibool\ false} \times \tau_1); S, \phi \vdash J \triangleright i_1\end{array}}{\Omega, \Lambda, R, \tau; \tau_1 + \tau_2; S, \phi \vdash \text{BrSum}; J \triangleright C_{\text{BrSum}} + \max(i_1, i_2)} \text{ BrSum}$$

$$\frac{\Omega, \Lambda \vdash \tau' :: *}{\Omega, \Lambda, \mathsf{ibool\ false}; \tau; S \vdash \text{Inj}_{\tau'} : \Omega, \Lambda, \tau + \tau'; S \triangleright C_{\text{Inj}}} \text{ InjL}$$

Figure 3-3: TiEVM typing rules (sequences and jumps)

and that the current state and the target's state precondition are equivalent. Total resource cost is the target's resource bounds plus the jump overhead.

There are three rules for conditional jumps: JumpI, JumpIEx, and BrSum. The first two are for the real instruction JumpI, and the third one is for pseudo-instruction BrSum. According to the EVM specification, JumpI uses the operand on the top of the stack as the jump target and the second operand as a condition. If the condition is true (i.e., nonzero), the execution jumps to the target; otherwise it continues forward. In TiEVM, depending on the type of the condition operand, there are two static semantics. If the condition is of type bool, the plain primitive Boolean type, the typing rule JumpI applies. It is almost the same as rule Jump, except that the code following the JumpI instruction, J, should also be typechecked, and the total resource costs of JumpI; J should be the max of the costs of the two branches (plus some overhead), since both could be executed. Note that both the branches will be run in the stack context S (or the equivalent S'), without information about the result of the condition test. This will be different from the next rule, JumpIEx.

Rule JumpIEx applies when the condition operand is of type ibool i, a Boolean type indexed by the index i of sort Bool. The difference from rule JumpI is that in the two branches, the result of the condition test is available on the stack. This is manifested by the $\exists\{i = \mathsf{true}\}.\mathsf{unit}$ ($\exists\{i = \mathsf{false}\}.\mathsf{unit}$ for the other branch) type on the top of the stack for the true branch. The branches need to do unpacking of the existential package to expose the information in their VC-checking contexts. This version of JumpI static semantics needs the stack to contain a value of type unit as the third operand; this is a technical requirement, resulting from the fact that runtime behavior of JumpI is the same regardless of its static semantics, so the space to hold the existential package must already be there on the stack (JumpI cannot do extra stack manipulation). This unit is turned into an existential package by JumpI, which does not involve extra runtime behavior since existential packages do not change the runtime representation of the value, as described in Section 3.3.

The other conditional jump instruction, BrSum, is a pseudo-instruction for branching on sum types (i.e. tagged unions). The reason such a branching cannot be imple-

$$\frac{\tau = \forall \overrightarrow{a : sk}.\ (\phi, R, S) \xrightarrow{i} \blacksquare \qquad \mathsf{split}(\overrightarrow{a : sk}) = (\Omega, \Lambda) \qquad G|\Omega, \Lambda, R, S, \phi \vdash J \triangleright i' \qquad \Omega \vdash i' \leq i \qquad \cdot, \cdot \vdash \tau :: *}{G \vdash \mathsf{block}\ \{\tau : J\}}\ \textsc{Block}$$

$$\frac{\begin{array}{c} H = \{l_n \mapsto \tau_n | n < N\} \qquad G = (H, L, T) \qquad \phi_0 = \{u \mapsto 0 | T(u) = \mathsf{vector}\ _\ \text{or}\ \mathsf{icell}\} \\ G \vdash \mathsf{block}\ \{\tau_n : J_n\}\ \text{for every}\ n \qquad G \vdash \mathsf{block}\ \{(\phi_0, \cdot, \cdot) \xrightarrow{i} \blacksquare : J_M\} \end{array}}{L, T \vdash \mathsf{prog}\ (\{l_n \mapsto \mathsf{block}\ \{\tau_n : J_n\} | n < N\}, J_M)}\ \textsc{Prog}$$

Figure 3-4: Typing rules for TiEVM programs and basic blocks

mented by a single JumpI instruction is that branching on sum types involves reading the tag and the data payload of the sum-type operand from the memory heap to the stack and then branching on the tag. Sum-type values are larger than a word and can only be stored on the heap, because the data payload can use up an entire word and there needs to be an extra bit for the tag[11]. And when the tag and data payload are on the stack, the type of the payload depends on the value of the tag. Such a dependency between two stack slots cannot be expressed in the current type system, at least not without intricate typing tricks. Rule BrSum is similar to rule JumpIEx. Within the branches, a value of type τ_1 (τ_2 respectively) is available on the stack. It is paired with an ibool, which is a residual artifact from the implementation of the pseudo-instruction.

To complete the picture of sum types, I also show here an introduction rule for sum types, rule InjL. Instruction Inj is also a pseudo-instruction, whose effect is that given a value of type ibool false and a value of type τ on top of the stack, it will package them into a value of type $\tau + \tau'$ (for any choice of τ'). A symmetrical rule also exists for constructing $\tau' + \tau$ given a value of type ibool true. Notice that constructing a sum-type value involves allocating two slots on the heap to store the tag and the data payload. That is why there is a heap-allocation cost of 2. The result is a pointer to the 2-word memory buffer.

3.4.3 Basic blocks and whole programs

A TiEVM program consists of an entry instruction sequence and a list of basic blocks each associated with a code label. Their typing rules are listed in Figure 3-4. A basic block (τ, J) is well-typed, according to rule Block, if the instruction sequence J is well-typed in a context derived from the block specification τ. To be a valid block specification, the type τ is required to be a code-pointer type wrapped by any amount of index or type polymorphism. I use $a : sk$ to mean either an index-sort binding or a type-kind binding. All contexts can be read off from such a type (I use $\mathsf{split}(\overrightarrow{a : sk})$ to mean splitting a mixed index/type context into an index context and a type context).

A TiEVM program $(J_M, \{l_n \mapsto (\tau_n, J_n) | n < N\})$ is well-typed, according to rule Prog, if each of the basic blocks (τ_n, J_n) is well-typed,[12] and the entry sequence J_M (the "main code") is well-typed under an empty context. The code-label typing context H used for checking the basic blocks and the main code is constructed by mapping each code label to its corresponding block's specification type. The main code is checked with an empty state u which maps every state name whose type is vector or indexed cell ("icell" for short) to zero (see Section 3.4.7 for states and storage operations). There is no specification for the costs of the main code (which are also the costs of the whole program[13]), so it is well-typed as long as its costs can be bounded by some bounds i (they still need to be bounded). We can use the AscTime and AscSpace instructions described in Section 3.4.8 as specifications for the program's costs.

3.4.4 Stack manipulation and simple arithmetic

Stack manipulation is done via four instructions: Push, Pop, Swap, and Dup, for pushing to the stack, popping from the stack, swapping two stack positions, and duplicating a stack cell (and putting the duplicate on the top). The semantics and typing rules for the last three are straightforward, and I will not discuss them here. The rule

[11] In the future I may choose to reserve one bit in primitive types to use for tagging.

[12] There could be zero basic blocks, when $N = 0$.

[13] Because cost specifications always refer to costs from now to the very end of the execution

$$\frac{H, L, T|\Omega, \Lambda \vdash w : \tau}{H, L, T|\Omega, \Lambda, S \vdash \textsc{Push}_n\ w : \Omega, \Lambda, \tau; S \rhd C_{\textsc{Push}}}\ \textsc{Push}$$

$$\frac{}{\vdash \overline{n} : \mathsf{nat}\ n}\ \textsc{Nat} \qquad \frac{H(l) = \tau}{H| \vdash l : \tau}\ \textsc{Label} \qquad \frac{\Omega, \Lambda \vdash \tau :: * \qquad \Omega \vdash \bot}{\Omega, \Lambda \vdash \mathsf{never}_\tau : \tau}\ \textsc{Never}$$

$$\frac{\tau_1 = \mathsf{nat}\ n_1 \qquad \tau_2 = \mathsf{nat}\ n_2 \qquad \tau = \mathsf{nat}\ (n_1 + n_2)}{\tau_1; \tau_2; S \vdash \textsc{Add} : \tau; S \rhd C_{\textsc{Add}}}\ \textsc{AddNat}$$

$$\frac{\tau_1 = \mathsf{tuplePtr}\ \overrightarrow{\tau_T}\ n_1\ b \qquad \tau_2 = \mathsf{nat}\ n_2 \qquad \tau = \mathsf{tuplePtr}\ \overrightarrow{\tau_T}\ (n_1 + n_2)\ b}{\tau_1; \tau_2; S \vdash \textsc{Add} : \tau; S \rhd C_{\textsc{Add}}}\ \textsc{AddTuple}$$

$$\frac{\tau_1 = \mathsf{arrayPtr}\ \tau'\ j\ i_1 \qquad \tau_2 = \mathsf{nat}\ i_2 \qquad \tau = \mathsf{arrayPtr}\ \tau'\ j\ (i_1 + i_2)}{\tau_1; \tau_2; S \vdash \textsc{Add} : \tau; S \rhd C_{\textsc{Add}}}\ \textsc{AddArray}$$

$$\frac{\tau_1 = \mathsf{nat}\ n_1 \qquad \tau_2 = \mathsf{nat}\ n_2 \qquad \tau = \mathsf{ibool}\ (n_1 <?\ n_2)}{\tau_1; \tau_2; S \vdash \textsc{Lt} : \tau; S \rhd C_{\textsc{Lt}}}\ \textsc{LtNat}$$

Figure 3-5: TiEVM typing rules (stack manipulations and simple arithmetic)

for PUSH, in Figure 3-5, relies on the word-value typing judgment $H, L, T|\Omega, \Lambda \vdash w : \tau$, for which I only show rules NAT, LABEL, and NEVER as examples. The word-value typing rule for state names is also of interest; I will discuss it together with storage operations in Section 3.4.7. never is only well-typed when False can be proved in the current index context, meaning that information gathered when typechecking previous code has proved that this part of the program will never be reached. This is similar to TiML's typing rule for never.

I use instruction ADD as an example for arithmetic operations. ADD is quite versatile in TiEVM; many things can be added, such as integers, indexed natural numbers, and all sorts of pointers. I show rules ADDNAT, ADDTUPLE, and ADDARRAY as examples. Which rule will apply depends on the types of the operands on the stack (i.e. instruction ADD is overloaded for different types). A pointer can be added to a number (usually of type nat i), resulting in another pointer with the offset field adjusted. Symmetric ADD rules exist to allow for swapping the left and right addends. Similar rules exist for the subtraction instruction SUB (a number can be subtracted from a pointer, but not the other way around).

$$\frac{\tau = \mathsf{tuplePtr}\ \overrightarrow{\tau_T}\ n\ \mathsf{false} \qquad n\ \mathsf{mod}\ 32 = 0 \qquad n/32 < |\overrightarrow{\tau_T}| \qquad \overrightarrow{\tau_T}(n/32) = \tau'}{\tau; S \vdash \text{MLoad} : \tau'; S \triangleright C_{\text{MLoad}}}\ \text{MLoadTuple}$$

$$\frac{\tau = \mathsf{arrayPtr}_{32}\ \tau'\ j\ i \qquad \Omega \vdash i\ \mathsf{mod}\ 32 = 0 \wedge 1 \leq i/32 \leq j}{\Omega, \tau; S \vdash \text{MLoad} : \Omega, \tau'; S \triangleright C_{\text{MLoad}}}\ \text{MLoadArray32}$$

$$\frac{\tau = \mathsf{arrayPtr}_1\ \tau'\ j\ i \qquad \Omega \vdash i \leq j}{\Omega, \tau; S \vdash \text{MLoad} : \Omega, \mathsf{int}; S \triangleright C_{\text{MLoad}}}\ \text{MLoadArray1}$$

$$\frac{\tau_1 = \mathsf{arrayPtr}_{32}\ \tau\ j\ i \qquad \Omega \vdash i\ \mathsf{mod}\ 32 = 0 \wedge 1 \leq i/32 \leq j \qquad \Omega, \Lambda \vdash \tau_2 \equiv \tau}{\Omega, \Lambda, \tau_1; \tau_2; S \vdash \text{MStore} : \Omega, \Lambda, S \triangleright C_{\text{MStore}}}\ \text{MStoreArray32}$$

$$\frac{\tau_1 = \mathsf{arrayPtr}_1\ \tau\ j\ i \qquad \Omega \vdash 32 \leq i < j + 32 \qquad \Omega, \Lambda \vdash \tau_2 \equiv \tau}{\Omega, \Lambda, \tau_1; \tau_2; S \vdash \text{MStore8} : \Omega, \Lambda, S \triangleright C_{\text{MStore8}}}\ \text{MStore8}$$

$$\frac{\tau = \mathsf{arrayPtr}_n\ \tau'\ j\ 0}{\tau; S \vdash \text{MLoad} : \mathsf{nat}\ j; S \triangleright C_{\text{MLoad}}}\ \text{MLoadArrayLen}$$

$$\frac{\tau = \mathsf{nat}\ m \qquad \mathsf{isRegAddr}(m) = n \qquad R(n) = \tau'}{R, \tau; S \vdash \text{MLoad} : R, \tau'; S \triangleright C_{\text{MLoad}}}\ \text{MLoadReg}$$

$$\frac{\tau_1 = \mathsf{nat}\ m \qquad \mathsf{isRegAddr}(m) = n}{R, \tau_1; \tau_2; S \vdash \text{MStore} : R[n \mapsto \tau_2], S \triangleright C_{\text{MStore}}}\ \text{MStoreReg}$$

Figure 3-6: TiEVM typing rules (memory access)

Other arithmetic operations such as Mul, Div, Mod, and Exp only apply to numbers (not pointers) and hence are simpler. The same is true for comparisons. Comparisons can be made between indexed natural numbers; the result is of type $\mathsf{ibool}\ \phi(i_1, i_2)$, where $\phi(i_1, i_2)$ is the index operation that corresponds to this comparison. An example, rule LtNat, is shown (<? is the less-than-test operator that returns true or false based on the comparison result).

3.4.5 Memory access

Values of types that are larger than a word are stored on memory (the “heap”). These include tuples, arrays, and sum types that have been discussed in Section 3.4.2. Access to tuples and arrays can only be done element-wisely, at word level; there are no operations for whole-tuple or whole-array moving or copying. Access to a tuple

component or array element usually consists of two steps: doing pointer arithmetic to move the pointer to the desired position, and then doing read/write using instruction MLoad/MStore.[14] Pointer arithmetic has been described in Section 3.4.4, so this section will focus on typing rules for MLoad and MStore, shown in Figure 3-6.

Each tuple must have at least two components, and each component occupies a word. Components are stored in memory continuously, so an n-tuple is stored in n consecutive words. These components can be pointers to other in-memory structures, enabling tuples of tuples or tuples of arrays. Tuples are read-only after initialization (so they can be treated as values in the functional-programming sense). Tuple and array initialization need special treatment and will be discussed in Section 3.4.6. The rule for using MLoad to read a tuple component is MLoadTuple. Note that MLoad is also overloaded for different types of operands, similar to Add. A tuple pointer type, $\mathsf{tuplePtr}\ \overrightarrow{\tau}\ n\ b$, as described in Section 3.3, contains the types of its components, the offset, and a Boolean flag indicating whether this is an on-storage tuple. The offset is a concrete natural number (as opposed to an index), so the typechecker can use it to retrieve the component type $\overrightarrow{\tau_T}(n/32)$ (components are word-sized i.e. 32-byte wide[15]). The offset is expressed in units of bytes because MLoad can read out a word at any byte position, according to the EVM specification. The on-storage flag is required to be false in this rule.

As in TiML, arrays in TiEVM are indexed by lengths so any access to an element will generate a VC checking that the target position is in-bounds. The array-pointer type, $\mathsf{arrayPtr}_n\ \tau\ j\ i$, contains the element width n (in bytes), the element type τ, the length j, and the offset i (remember that TiML's array type is $\mathsf{array}_n\ \tau\ j$, without the offset). The offset of element m (the $(m+1)$-th element) is $32+n\times m$, because offsets 0 to 31 are used to store the length (in order to support the "get length" operation at runtime, as described below). That is why the VC is $\Omega \vdash i\ \mathsf{mod}\ 32 = 0 \wedge 1 \leq i/32 \leq j$ in rule MLoadArray32, the rule for reading a 32-byte array. For reading a 1-byte array, because EVM does not provide an instruction for reading just 1 byte, we need

[14]The M- prefix stands for "memory," as opposed to the "storage" operations discussed in Section 3.4.7.

[15]Note that in EVM words are unusually large, 512-bit (or 32-byte) wide.

to read a 32-byte chunk by MLoad and then use Int2Byte or Int2Bool (whose typing rules are omitted) to get the rightmost byte. The VC $i \leq j$ is enough to guarantee the safety of accessing addresses $a+i, \cdots, a+i+31$ (assuming the array starts at address a), because the array spans from address a to $a+32+j$ (including the length field). The type system currently only allows reading/writing 1-byte and 32-byte arrays.

Rule MStoreArray32, for writing to 32-byte arrays, is similar to rule MLoadArray32. The operand on top of the stack is the array pointer and the second is the value to write. Array reading with the offset 0 is treated as the "get length" operation, indicated by rule MLoadArrayLen. In this case the result type is $\mathsf{nat}\ j$, j being the length. Writing to 1-byte arrays is done by MStore8, the EVM instruction for just overwriting one byte.

Registers

MLoad and MStore are also used to implement registers. The purpose of registers is to implement local variables for higher-level languages[16]. Any memory access at an explicit address (as a compile-time-known natural number) that is lower than $32 \times \mathsf{NumRegs}$ and a multiple of 32 is treated as accessing a register (multiple of 32 because registers are word-sized). The parameter NumRegs can be determined at compile time in a whole-program compilation. Therefore, the $(n+1)$-th register, r_n (read as "register n"), is accessed by doing MLoad and MStore at address $32 \times n$. Registers lower than FirstGeneralReg (currently 3) are reserved as scratch space for some pseudo-instructions and are forbidden to be accessed in the type system.

Register reading and writing are governed by rules MLoadReg and MStoreReg. These rules apply when the operand on stack is of type $\mathsf{nat}\ m$ instead of a tuple or array pointer. I write m to mean that the index must be a concrete natural number known at the time of typechecking (containing no symbolic part). A test function $\mathsf{isRegAddr}(m)$ is used to see whether address m is a valid register address

[16] Local variables are never put on the stack. When they need to be moved out of registers to make room for the callee function, they are put into the closure of the continuation. See Chapter 4 for details.

$$\frac{\Omega, \Lambda \vdash \tau :: * \text{ for every } \tau \text{ in } \overrightarrow{\tau} \qquad n = |\overrightarrow{\tau}|}{\Omega, \Lambda, S \vdash \textsc{TupleMalloc}_{\overrightarrow{\tau}} : \Omega, \Lambda, \mathsf{preTuple}\ \overrightarrow{\tau}\ 0\ n; S \triangleright C_{\textsc{TupleMalloc}}(n)} \textsc{TupleMalloc}$$

$$\frac{\tau_2 = \mathsf{preTuple}\ \overrightarrow{\tau}\ n\ m \qquad n \ \mathsf{mod}\ 32 = 0 \qquad n/32 + 1 = m \qquad \Omega, \Lambda \vdash \tau_1 \equiv \overrightarrow{\tau}(m-1)}{\Omega, \Lambda, \tau_1; \tau_2; S \vdash \textsc{TupleInit} : \Omega, \Lambda, \mathsf{preTuple}\ \overrightarrow{\tau}\ n\ (m-1); S \triangleright C_{\textsc{TupleInit}}} \textsc{TupleInit}$$

$$\frac{}{\mathsf{preTuple}\ \overrightarrow{\tau}\ n\ 0; S \vdash \textsc{TupleDone} : \mathsf{tuplePtr}\ \overrightarrow{\tau}\ n; S \triangleright 0} \textsc{TupleDone}$$

$$\frac{\Omega, \Lambda \vdash \tau :: * \qquad \tau_1 = \mathsf{nat}\ i \qquad j = b\ ?\ 0 : i}{\Omega, \Lambda, \tau_1; S \vdash \textsc{ArrayMalloc}^{n,b}_{\tau} : \Omega, \Lambda, \mathsf{preArray}_n\ \tau\ i\ j\ \mathsf{false}\ b; S \triangleright C_{\textsc{ArrayMalloc}}(i, n)} \textsc{ArrayMalloc}$$

$$\frac{\tau_1 = \mathsf{nat}\ i \qquad \tau_2 = \mathsf{preArray}_n\ \tau\ i_{\mathsf{len}}\ j\ b\ \mathsf{false} \qquad \Omega, \Lambda \vdash \tau_3 \equiv \tau \qquad \Omega \vdash i \ \mathsf{mod}\ n = 0 \land i/n + 1 = j}{\Omega, \Lambda, \tau_1; \tau_2; \tau_3; S \vdash \textsc{ArrayInit}_n : \Omega, \Lambda, \tau_1; \mathsf{preArray}_n\ \tau\ i_{\mathsf{len}}\ (j-1)\ b\ \mathsf{false}; S \triangleright C_{\textsc{ArrayInit}}} \textsc{ArrayInitDown}$$

$$\frac{\tau_1 = \mathsf{nat}\ i \qquad \tau_2 = \mathsf{preArray}_n\ \tau\ i_{\mathsf{len}}\ j\ b\ \mathsf{true} \qquad \Omega, \Lambda \vdash \tau_3 \equiv \tau \qquad \Omega \vdash i \ \mathsf{mod}\ n = 0 \land i/n = j}{\Omega, \Lambda, \tau_1; \tau_2; \tau_3; S \vdash \textsc{ArrayInit}_n : \Omega, \Lambda, \tau_1; \mathsf{preArray}_n\ \tau\ i_{\mathsf{len}}\ (j+1)\ b\ \mathsf{true}; S \triangleright C_{\textsc{ArrayInit}}} \textsc{ArrayInitUp}$$

$$\frac{\tau_1 = \mathsf{nat}\ i' \qquad \tau_2 = \mathsf{preArray}_n\ \tau\ i\ j\ _\ b \qquad \Omega \vdash i' = i}{\Omega, \Lambda, \tau_1; \tau_2; S \vdash \textsc{ArrayInitLen} : \Omega, \Lambda, \mathsf{preArray}_n\ \tau\ i'\ j\ \mathsf{true}\ b; S \triangleright C_{\textsc{ArrayInitLen}}} \textsc{ArrayInitLen}$$

$$\frac{\tau_1 = \mathsf{preArray}_n\ \tau\ i\ j\ \mathsf{true}\ b \qquad \Omega \vdash j = b\ ?\ i : 0}{\Omega, \tau_1; S \vdash \textsc{ArrayDone} : \Omega, \mathsf{arrayPtr}_n\ \tau\ i\ 32; S \triangleright 0} \textsc{ArrayDone}$$

Figure 3-7: TiEVM typing rules (tuple and array initialization)

and if so what the register number is. It is a partial function defined as $m/32$ if $m \ \mathsf{mod}\ 32 = 0$ and $\mathsf{FirstGeneralReg} \leq m/32 < \mathsf{NumRegs}$. The type of the current value stored in a register is retrieved using the register number from the register typing context R. It is a partial map from register numbers to types. Note in rule MStoreReg that a register write will usually change the type for that register, hence register updates are “strong updates” in the sense that the type can be changed.

3.4.6 Tuple and array initialization

In a high-level language such as TiML, a tuple is constructed in a single step, and an array is allocated with every element set to an initial value also in a single step. But at the TiEVM level, the initialization of tuples and arrays needs the cooperation of multiple instructions. A pseudo-instruction could have been added to initialize a

tuple in one step, but for an array, because its length may only be known at runtime, setting all its elements needs a mini for-loop, for which multiple basic blocks need to be generated. I chose to break up the initialization process for both tuples and arrays into three stages: (1) allocating the memory space; (2) assigning to each element, word by word; and (3) using a noop instruction (zero-cost instruction) to signal to the typechecker that the initialization process has finished. During the initialization, special types need to be used to regulate operations on the half-constructed structure[17].

For tuples, these three stages are governed by typing rules TupleMalloc, TupleInit, and TupleDone in Figure 3-7. In TiEVM, heap allocation is implemented by using the first word in the scratch space (at address 0) as the free pointer, pointing to the top of the current heap, which is where the next available free space is. The pseudo-instruction TupleMalloc increases the free pointer by the size of the new tuple and returns the old value of the free pointer. The returned pointer is given the type $\mathsf{preTuple}\ \overrightarrow{\tau}\ 0\ n$, which represents a pointer to a tuple in the making. $\overrightarrow{\tau}$ and 0 are the component types and the offset, akin to type $\mathsf{tuplePtr}$. The last field, n, is called the "lowest initialized position." This is an indicator for recording which of the tuple's components have been initialized. In a more general design, an n-bit flag could be used to allow an arbitrary subset of components to be initialized; but in this thesis, since the compiler-generated code always sets the tuple components from last to first, I just need a number to indicate progress. At start, the lowest initialized position is n, meaning that none of the components have been initialized (components are numbered 0 to $n-1$). When the initialization has finished, this number needs to be 0, as required by rule TupleDone. Writing to a component can only be done at position $m-1$, where m is the lowest initialized position (in units of words), as manifested by the premises in rule TupleInit (the offset n is in units of bytes). After the TupleInit instruction, the lowest initialized position is decreased

[17]TAL [63] handled the initialization of complex values in a similar way by introducing new types for half-constructed values. It even dedicated a compilation phase to inserting the initialization steps. I choose to merge this phase into the code-generation phase. See Chapter 4 for more details.

by one.[18]

Array initialization, starting from rule ArrayMalloc, is regulated by type $\mathsf{preArray}_n\ \tau\ i\ j\ b_1\ b_2$. Here, n is the element width, i is the length of the new array, and j is the progress marker akin to the "lowest initialized position" above. There are two extra Boolean flags in this type: b_1 indicates whether the length data has been initialized, and b_2 indicates whether the initialization direction is "upward," from low positions to high ones. Arrays could be initialized in both upward and downward direction, both used by the compiler[19]. The length data, stored at the beginning of the array's memory space, is initialized by a separate pseudo-instruction from ArrayMalloc, as specified by rule ArrayInitLen[20]. In the downward element-assigning step, governed by rule ArrayInitDown, the progress marker j has the same meaning as the "lowest initialized position" for tuples. Because j is an index instead of a concrete number, the requirement on its value is expressed as a VC. In the upward direction governed by rule ArrayInitDown, j means "highest uninitialized position," and the VC is changed accordingly. The direction is chosen at allocation time, determined by instruction ArrayMalloc's annotation. Initialization completion is governed by rule ArrayDone, checking that both the length and all the elements have been set. The resulting arrayPtr pointer points to the first element of the array; the offset 32 is for skipping the length data.

3.4.7 Storage access

Storage access is performed via a different set of instructions from those in Section 3.4.5. They are listed in Figure 3-8. I will first describe the organization of the flat storage space in EVM into different kinds of data containers, and then discuss operations on each of these containers.

[18] TupleInit cannot be just MStore, because TupleInit changes the type of the pointer by modifying the lowest-initialized-position flag.

[19] One for initializing an array with the same element value and the other for initializing an array using a list of values

[20] Pseudo-instruction ArrayInitLen could have been merged into instruction ArrayMalloc. I chose to keep pseudo-instructions simpler and single-purpose, if that does not complicate the type system too much.

$$\frac{L(n) = u \qquad u \in \mathsf{dom}(T)}{L, T \vdash \mathsf{state}\ n : \mathsf{state}\ u}\ \textsc{State} \qquad \frac{L(n) = u \qquad T(u) = \tau \qquad \tau = \mathsf{map}\ _}{L, T \vdash \mathsf{state}\ n : \mathsf{tuplePtr}\ [\tau]\ 0\ \mathsf{true}}\ \textsc{StateMap}$$

$$\frac{L(n) = u \qquad T(u) = \mathsf{cell}\ \tau}{L, T \vdash \mathsf{state}\ n : \mathsf{tuplePtr}\ [\tau]\ 0\ \mathsf{true}}\ \textsc{StateCell}$$

$$\frac{\tau = \mathsf{tuplePtr}\ \overrightarrow{\tau_T}\ n\ \mathsf{true} \qquad n < |\overrightarrow{\tau_T}| \qquad \overrightarrow{\tau_T}(n) = \tau' \qquad \mathsf{isWordsizeType}(\tau')}{\tau; S \vdash \text{SLoad} : \tau'; S \triangleright C_{\text{SLoad}}}\ \textsc{SLoadTuple}$$

$$\frac{\tau_1 = \mathsf{tuplePtr}\ \overrightarrow{\tau_T}\ n\ \mathsf{true} \qquad n < |\overrightarrow{\tau_T}| \qquad \overrightarrow{\tau_T}(n) = \tau \qquad \mathsf{isWordsizeType}(\tau) \qquad \Omega, \Lambda \vdash \tau_2 \equiv \tau}{\Omega, \Lambda, \tau_1; \tau_2; S \vdash \text{SStore} : \Omega, \Lambda, S \triangleright C_{sset}}\ \textsc{SStoreTuple}$$

$$\frac{\tau = \mathsf{state}\ u \qquad T(u) = \mathsf{icell} \qquad \phi(u) = i}{T|\tau; S, \phi \vdash \text{SLoad} : \mathsf{nat}\ i; S, \phi \triangleright C_{\text{SLoad}}}\ \textsc{SLoadICell}$$

$$\frac{\tau_1 = \mathsf{state}\ u \qquad T(u) = \mathsf{icell} \qquad \phi(u) = i_1 \qquad \tau_2 = \mathsf{nat}\ i_2}{T|\tau_1; \tau_2; S, \phi \vdash \text{SStore} : S, \phi[u \mapsto i_2] \triangleright i_1 = 0 \wedge i_2 \neq 0\ ?\ C_{sset} : C_{sreset}}\ \textsc{SStoreICell}$$

$$\frac{\tau = \mathsf{vectorPtr}\ u\ i \qquad T(u) = \mathsf{vector}\ \tau' \qquad \phi(u) = j \qquad \Omega \vdash i < j}{T|\Omega, \tau; S, \phi \vdash \text{SLoad} : \Omega, \tau'; S, \phi \triangleright C_{\text{SLoad}}}\ \textsc{SLoadVector}$$

$$\frac{\tau_1 = \mathsf{vectorPtr}\ u\ i \qquad T(u) = \mathsf{vector}\ \tau \qquad \phi(u) = j \qquad \Omega \vdash i < j \qquad \Omega, \Lambda \vdash \tau_2 \equiv \tau}{T|\Omega, \Lambda, \tau_1; \tau_2; S, \phi \vdash \text{SStore} : \Omega, \Lambda, S, \phi \triangleright C_{sset}}\ \textsc{SStoreVector}$$

$$\frac{\tau = \mathsf{state}\ u \qquad T(u) = \mathsf{vector}\ _ \qquad \phi(u) = i}{T|\tau; S, \phi \vdash \text{SLoad} : \mathsf{nat}\ i; S, \phi \triangleright C_{\text{SLoad}}}\ \textsc{SLoadVectorLen}$$

$$\frac{\tau_1 = \mathsf{state}\ u \qquad T(u) = \mathsf{vector}\ _ \qquad \tau_2 = \mathsf{nat}\ i \qquad \Omega \vdash i = 0}{T|\Omega, \tau_1; \tau_2; S, \phi \vdash \text{SStore} : \Omega, S, \phi[u \mapsto 0] \triangleright C_{srest}}\ \textsc{VectorClear}$$

$$\frac{\tau_2 = \mathsf{state}\ u \qquad T(u) = \mathsf{vector}\ \tau \qquad \phi(u) = i \qquad \Omega, \Lambda \vdash \tau_1 \equiv \tau}{T|\Omega, \Lambda, \tau_1; \tau_2; S, \phi \vdash \textsc{VectorPushBack} : \Omega, \Lambda, S, \phi[u \mapsto i + 1] \triangleright C_{\textsc{VectorPushBack}}}\ \textsc{VectorPushBack}$$

$$\frac{\tau = \mathsf{tuplePtr}\ \overrightarrow{\tau_T}\ n\ \mathsf{true} \qquad n < |\overrightarrow{\tau_T}| \qquad \overrightarrow{\tau_T}(n) = \mathsf{map}\ (\mathsf{tuple}\ \overrightarrow{\tau_V})}{\mathsf{int}; \tau; S \vdash \textsc{MapPtr} : \mathsf{tuplePtr}\ \overrightarrow{\tau_V}\ 0\ \mathsf{true}; S \triangleright C_{\textsc{MapPtr}}}\ \textsc{MapPtr}$$

$$\frac{\tau_1 = \mathsf{state}\ u \qquad \tau_2 = \mathsf{nat}\ i}{\tau_1; \tau_2; S \vdash \textsc{VectorPtr} : \mathsf{vectorPtr}\ u\ i; S \triangleright C_{\textsc{VectorPtr}}}\ \textsc{VectorPtr}$$

$$\frac{\tau_1 = \mathsf{tuplePtr}\ \overrightarrow{\tau}\ n\ \mathsf{true} \qquad n + m \leq |\overrightarrow{\tau}| \qquad \overrightarrow{\tau}(n, \cdots, n + m - 1) = \overrightarrow{\tau_R}}{\tau_1; S \vdash \textsc{RestrictView}_m : \mathsf{tuplePtr}\ \overrightarrow{\tau_R}\ 0\ \mathsf{true}; S \triangleright 0}\ \textsc{RestrictView}$$

Figure 3-8: TiEVM typing rules (storage access)

Storage organization

The storage model EVM provides is just a big flat map from words to words (for each account). In contrast, TiEVM provides four types of storage containers: maps, vectors, cells, and indexed cells. Maps map words to tuples, whose components can be words or other maps. Maps of maps are useful in smart contracts to store values indexed by two keys. Map codomains (the types of map values) must be inhabited by zero (or a value whose bit representation is all-zero), due to the implementation of maps on top of EVM's native storage, as discussed below. Vectors contain elements of any word-sized type (not required to be inhabited by zero). If a function needs to access a vector, the vector's length must be specified in the function's pre- and post-condition, and accesses are bounds-checked statically. A cell can be seen as a vector whose length is always one and thus not needed to be specified in pre/postconditions. An indexed cell ("icell" for short) contains a natural number whose value is specified in pre/postconditions. Vectors and icells are used when a function's cost depends on the current size or value of some on-storage data; since such information is not related to the function's arguments, the function's cost must be specified in terms of the figures in its precondition, corresponding to vector lengths and icell values.

The magic of using a single flat map to implement multiple instances of different kinds of containers is borrowed from Solidity [8], based on hash-function tricks. Technically, each map is associated with a "base address" on storage. There is nothing stored on that address; it is only used as a unique identifier for each map. The address of the value associate with key k (a word) in the map identified by base address m is calculated by $\mathsf{sha3}(k \cdot m)$, where $\mathsf{sha3}$ is the Keccak-256 hash function mapping an arbitrary bit sequence to a word, and $k \cdot m$ is the concatenation of word k and word m to form a 1024-bit (two-word) sequence. Such details are hidden by the pseudo-instruction MapPtr and not visible in TiEVM's type system. The base address of a map is determined as follows: if the map is a standalone state with a state name, its base address is the integer encoding of the state name (currently the rank of its appearance); otherwise, if the map is a value (or a component of a tuple value) of

another map, the base address is the address of the value (the address of a map value can always be calculated, using the $\mathsf{sha3}(k \cdot m)$ formula). Therefore, to read the value of $m[k_1].3[k_2]$ for example, which means retrieving with key k_1, projecting out the third component which is another map, and then retrieving with key k_2, we can first calculate the target address using $\mathsf{sha3}(\mathsf{k}_2 \cdot (\mathsf{sha3}(\mathsf{k}_1 \cdot \mathsf{m}) + 64))$, and then read at that address[21]. Readers are referred to the online Solidity documentation (under Section "Solidity in Depth > Miscellaneous") for a good description of the mechanism. Each vector is also associated with a base address. The address of the first element of vector v is calculated by $\mathsf{sha}(v)$; elements are stored continuously following that address. The value stored at the base address itself is the length of the vector. Vectors need this sha3 hashing because vectors can be enlarged and shrunk at runtime, so different vectors should be placed at far-away addresses on storage that have little chance of collision.

Typing rules for storage operations

Storage operations interact closely with TiEVM's state mechanism, so they are discussed together in this section. The first three rules in Figure 3-8, State, StateMap, and StateCell, are word-value typing rules governing how to type the word-value state n. In the most general case, the type of state n is type state u, where u is the state name encoded by n. Map L is used to retrieve the state name by its encoding. The state typing context, T, gives the type of a state name. Rule State only applies if rules StateMap and StateCell do not. These latter two rules say that if the state's type is a map or a cell, the result of the word-value state n is a tuple pointer pointing to an on-storage tuple. These two rules can as seen as "eager" to determine types, while rule State is being "lazy" by using a placeholder type state u as a temporary solution and delegating the final resolution of u to other rules. The reason for using lazy type resolution for vectors and icells is explained when I discuss vector operations.

[21] 64 because the address of the third component is $(3 - 1) \times 32 = 64$. The address of the first component is 0. Each word is 32 bytes wide.

The types of states belong to the same syntax class, τ, as regular types, but they should be understood as a special set of types only used to describe the storage containers. They have their own well-formedness rules, checked when typechecking state declarations. Briefly, maps always use words as keys, and their values are tuples of word-sized types or other maps (or mixtures of them). One-component tuples are allowed (as opposed to in-memory tuples, which must have at least two components). A vector can only have word-sized elements of the same type. There are no vectors of maps[22] or vectors of vectors. A cell can contain a word-sized value, of a fixed type. An icell always contains a natural number. Word-sized values in this section mean word-sized values that are meaningful when stored on storage; they include primitive types but do not include pointers to in-memory structures.

On-storage tuples are readable and writable at each component, as specified by rule SLoadTuple and SStoreTuple. The former is similar to rule MLoadTuple for in-memory tuple read, except that the on-storage flag in the tuple pointer type $\mathsf{tuplePtr}\ \overrightarrow{\tau_T}\ n\ \mathsf{true}$ is required to be true here. SLoad and SStore are real instructions overloaded for different operand types, similar to MLoad and MStore. $\mathsf{isWordsizeType}(\tau')$ is checked here to rule out the case when τ' is $\mathsf{map}\ _$ (in which case rule MapPtr should be used). The cost of a tuple write, says rule SStoreTuple, is C_{sset}. I will discuss the subtlety of storage-write cost when I describe rule SStoreICell below. Note that the type of a cell is $\mathsf{tuplePtr}\ [\tau]\ 0\ \mathsf{true}$, according to rule StateCell, so reading and writing of cells are also governed by SLoadTuple and SStoreTuple (cells are essentially one-component tuples).

When reading an icell, governed by rule SLoadICell, an indexed natural number of type $\mathsf{nat}\ i$ is returned. The index i is obtained from the current state ϕ via the state name u. When writing to an icell with a new indexed natural number, the state is updated with the new value, $\phi[u \mapsto i_2]$, as can be seen in rule SStoreICell. The use of the state name u explains why I chose to use $\mathsf{state}\ u$ as the type of an icell (instead of directly using icell): I need the state name to look up in and update the current state. The current state could have been changed between the typing of

[22] They could be added in the future.

$\mathsf{state}\ n$ and the actual SLOAD. The same reasoning also applies for vectors.

The cost of writing an icell is $i_1 = 0 \wedge i_2 \neq 0\ ?\ C_{sset} : C_{sreset}$.[23] This subtlety comes from EVM's cost specification, which says that the cost is C_{sset} (currently 20000) when one changes a storage cell from zero to a nonzero value and C_{sreset} (currently 5000) otherwise. This policy is for incentivizing smart contracts to keep their storage footprints as small as possible, since larger storage footprints translate to larger disk space required to run an Ethereum node. This poses a difficulty for TiEVM to estimate storage-write costs accurately. In the case of icells, TiEVM has enough information to determine the exact cost; but in cases such as tuples, TiEVM can only use the larger cost C_{sset} as a conservative estimate. This is a major source of estimation inaccuracy. I will discuss more about this issue in Chapter 5.

Reads and writes on elements of vectors are governed by rules SLoadVector and SStoreVector. These rules apply when the top-of-stack operand is $\mathsf{vectorPtr}\ u\ i$, the type of vector pointers. I will describe how to get a value of this type when discussing rule VectorPtr. Vector reading and writing are similar to the corresponding array operations described in Section 3.4.5, with the same boundary checks. The index i in vector-pointer type $\mathsf{vectorPtr}\ u\ i$ stands for the pointer offset, not the length of the vector; the length j is looked up in the current state ϕ via the state name u: $\phi(u) = j$. As with tuple writes, there is not enough information to determine the zeroness statically of the old and new value of the target vector element in rule SStoreVector, so the conservative cost C_{sset} is used.

Vectors have another three operations, governed by rules SLoadVectorLen, VectorPushBack, and VectorClear, for getting the length of the vector, appending an element at the end of the vector, and emptying the vector by setting its length to zero. Reading and resetting length are performed by the same SLoad and SStore instructions as before. Rules SLoadVectorLen and VectorClear apply when the first operand is of type $\mathsf{state}\ u$ instead of $\mathsf{vectorPtr}$. Both VectorClear and VectorPushBack change the length of the vector, so the state ϕ needs to be updated with the new length. Note that the cost in VectorClear is C_{sreset},

[23] The $i\ ?\ i_1 : i_2$ notation is the if-then-else expression for an index. See Chapter 2.

since we can be sure in this case the new value is zero.

I have described in the "storage organization" sector above how addresses of map values and vector elements are calculated using the sha3 hash trick. These calculations are implemented and hidden by the pseudo-instructions MapPtr and VectorPtr. As can be seen in rule MapPtr, from the type system's point of view, what instruction MapPtr does is that given a map key of type int (a word) and a pointer to an on-storage tuple, if the pointer points to a component that happens to be of type map (tuple $\overrightarrow{\tau_V}$)[24], MapPtr will construct a pointer pointing to the head of a tuple $\overrightarrow{\tau_V}$: tuplePtr $\overrightarrow{\tau_V}$ 0 true. This resulting pointer is the address of the value associated with the key given as the first operand. The second operand is the "base address" of the map as described in the "storage organization" sector. The base address is typed as a tuple pointer. Refer back to rule StateMap saying that the type of a map state is a tuple pointer.[25] The VectorPtr pseudo-instruction works similarly; given a first operand which is the base address of the vector and a second operand which is the number of the desired element (the offset), it constructs a pointer of type vectorPtr u i, pointing to the address of element i of the vector.

The last rule, RestrictView ("restricting the view of a pointer"), addresses the issue that different tuple-pointer types can represent the same storage address. For example, the address of type tuplePtr $[\tau_1, \tau_2, \tau_3]$ 1 true can also be typed tuplePtr $[\tau_2, \tau_3]$ 0 true and tuplePtr $[\tau_2]$ 0 true, among others. Rule RestrictView converts a tuple pointer pointing to the middle of a larger tuple to a pointer pointing to the beginning of a smaller tuple. The smaller tuple must be contained within the larger one. Instruction RestrictView is a noop instruction; it is used to implement tuples of tuples on storage at the source-language level, where they are flattened into single large tuples by the compiler. Instruction RestrictView corresponds to going from the outer tuple to the inner tuple on the source level.[26] Without this instruction, a tuple pointer with a larger view cannot, for example, be passed to a function expecting a

[24]Map values are always tuples.

[25]I essentially used tuple pointers to represent storage addresses.

[26]Readers familiar with C can think of it as dereferencing a 2D-array pointer, which does not change the pointer value but does change its type so that later operations such as "plus one" have different interpretations.

$$\frac{\tau_1 = \forall \alpha :: \kappa.\ \tau' \qquad \Omega, \Lambda \vdash \tau :: \kappa}{\Omega, \Lambda, \tau_1; S \vdash \text{APPT}_\tau : \Omega, \Lambda, \tau'[\tau/\alpha]; S \triangleright 0} \text{ APPT} \qquad \frac{\tau_1 = \forall a : s.\ \tau \qquad \Omega \vdash i : s}{\Omega, \tau_1; S \vdash \text{APPI}_i : \Omega, \tau[i/a]; S \triangleright 0} \text{ APPI}$$

$$\frac{\Omega, \Lambda \vdash \tau_1 :: * \qquad \tau_1 = \exists \alpha :: \kappa.\ \tau' \qquad \Omega, \Lambda \vdash \tau_2 :: \kappa \qquad \Omega, \Lambda \vdash \tau \equiv \tau'[\tau_2/\alpha]}{\Omega, \Lambda, \tau; S \vdash \text{PACK}^{\tau_1}_{\tau_2} : \Omega, \Lambda, \tau_1; S \triangleright 0} \text{ PACK}$$

$$\frac{\Omega, \Lambda \vdash \tau_1 :: * \qquad \tau_1 = \exists a : s.\ \tau' \qquad \Omega \vdash i : s \qquad \Omega, \Lambda \vdash \tau \equiv \tau'[i/a]}{\Omega, \Lambda, \tau; S \vdash \text{PACKI}^{\tau_1}_i : \Omega, \Lambda, \tau_1; S \triangleright 0} \text{ PACKI}$$

$$\frac{\Omega, \Lambda \vdash \tau :: * \qquad \tau = \tau_2\ \overrightarrow{c} \qquad \tau_2 = \mu \alpha :: \kappa.\ \tau_3 \qquad \Omega, \Lambda \vdash \tau_1 \equiv \tau_3[\tau_2/\alpha]\ \overrightarrow{c}}{\Omega, \Lambda, \tau_1; S \vdash \text{FOLD}_\tau : \Omega, \Lambda, \tau; S \triangleright 0} \text{ FOLD}$$

$$\frac{\tau = \tau_1\ \overrightarrow{c} \qquad \tau_1 = \mu \alpha :: \kappa.\ \tau_2}{\tau; S \vdash \text{UNFOLD} : \tau_2[\tau_1/\alpha]\ \overrightarrow{c}; S \triangleright 0} \text{ UNFOLD} \qquad \frac{\Omega, \Lambda \vdash \tau :: * \qquad \Omega, \Lambda \vdash \tau_1 \equiv \tau}{\Omega, \Lambda, \tau_1; S \vdash \text{ASCTYPE}_\tau : \Omega, \Lambda, \tau; S \triangleright 0} \text{ ASCTYPE}$$

$$\frac{\tau = \exists \alpha :: \kappa.\ \tau'}{\Lambda, \tau; S \vdash \text{UNPACK}_\alpha : \Lambda; \alpha :: \kappa, \tau'; S \triangleright 0} \text{ UNPACK} \qquad \frac{\tau = \exists a : s.\ \tau'}{\Omega, \tau; S \vdash \text{UNPACKI}_a : \Omega; a : s, \tau'; S \triangleright 0} \text{ UNPACKI}$$

$$\frac{\Omega = \Delta.1 \qquad \Omega \vdash i : \mathsf{Time} \qquad \Delta \vdash J \triangleright (i', j) \qquad \Omega \vdash i' \le i}{\Delta \vdash \text{ASCTIME}_i; J \triangleright (i, j)} \text{ ASCTIME}$$

Figure 3-9: TiEVM typing rules (miscellaneous)

pointer with a smaller view.

3.4.8 Miscellaneous

Figure 3-9 lists some of the other typing rules not covered in previous sections. They all have counterparts in TiML's typing rules and do not interact much with TiEVM's assembly-level features. The rules from APPT to UNPACKI, which are all noop instructions with zero costs, just change the type of the top-of-stack value. Readers can refer to the corresponding TiML typing rules in Section 2 for understanding the modifications of the type. The ASCTIME rule[27] relaxes the time estimation of the following instruction sequence, given that the provided new estimation can be proved to be no less than the original one. There is a similar ASCSPACE instruction for heap-allocation estimation. Note that in TiEVM every mention of resource costs means the costs from that point to the very end of the execution.

[27] "Asc" stands for "ascribe".

Chapter 4

The type-preserving compiler

The type-preserving compiler that compiles TiML programs to TiEVM programs consists of four major compilation phases (or "stages"): (1) a phase that translates a surface version of TiML (called Surface-TiML) to the TiML version that is described in Chapter 2 (called μTiML in this chapter); (2) a CPS-conversion phase that translates μTiML programs in direct style into μTiML programs in continuation-passing style; (3) a closure-conversion phase that makes the capturing of variables in locally defined functions explicit and turns each locally defined function into a top-level definition; (4) a code-generation phase that translates μTiML programs into TiEVM programs, taking care of register allocation and array/tuple initialization. All these phases assume that the input program is well-typed and guarantee that the output program be well-typed. Each of these four major phases will be covered by a section in this chapter.[1]

Besides these major phases, there are some minor phases including parsing, elaboration that translates an abstract-syntax tree (AST) used by the parser to an AST of Surface-TiML, name resolution that translates string representation of bound names into de Bruijn indices, pseudo-instruction expansion for TiEVM programs, and assembling of TiEVM programs that translates a TiEVM AST into a binary format (and takes care of code-label renaming). These minor phases do not involve typechecking

[1]Phases (2) and (3) are heavily influenced by [63], which in turn was influenced by the compilation strategy of SML/NJ [15].

and will not be discussed in this dissertation.

The main job of the compiler is to transform programs in the "top-down" direction, from the source language to the target language. But a special concern in this resource-analysis setting is that the compiler's behavior influences the costs of each operation in the source language (i.e. the cost model of the source language). The only ground truth about costs is the cost model of the target TiEVM language, given by the official EVM specification. Therefore, when designing and implementing the compiler, I needed to analyze its influence on costs and define the cost model for each intermediate language[2] in the "bottom-up" direction, firstly defining it for the lowest intermediate language and then moving up, until I reached a definition for the cost model of the Surface-TiML language. The last section in this chapter will describe these derived cost models.

4.1 Surface-TiML to μTiML

Section 2 described TiML as a System-F-like calculus that has building blocks such as recursive types, existential types, and sum types. Astute readers may have found that there is a discrepancy between this formulation and the code examples shown in Section 2.1, the latter of which use language features such as datatypes, pattern matching, and the `absidx` keyword. These code examples are actually written in a language called "Surface-TiML,"[3] and the formally defined language in Chapter 2 is called "μTiML" (read as "micro-TiML," as in "a core calculus for TiML"). Outside this section (and Section 4.5), I generally use "TiML" as a shorthand for μTiML, ignoring the fact that there is a higher-level language above it[4]. This section will describe Surface-TiML and the translation from it to μTiML.

[2]Phases (2), (3), and (4) share the same intermediate language, μTiML, as input but use different cost models.

[3]I apologize for the lack of a better name.

[4]μTiML is easier to define, prove sound, and use as an intermediate language, which is why this thesis focuses on μTiML most of the time. Surface-TiML is easier to actually program in for the user.

Type

τ ::= <same as μTiML minus $\mu\alpha : \kappa.\ \tau$, $\exists a : s.\ \tau$, and $\tau + \tau$>

| $\textsf{datatype}\ \alpha\ \overrightarrow{\beta}\ \overrightarrow{s}\ \textsf{as}\ \overrightarrow{c : \forall \overrightarrow{a : s}.\ \tau \to \alpha\ \overrightarrow{\beta}\ \overrightarrow{i}}$

Term

e ::= <same as μTiML minus let and irrelevant forms>

| $\textsf{case}\ e\ \textsf{of}\ \overrightarrow{p \Rightarrow e} \mid c\ \overrightarrow{\tau}\ \overrightarrow{i}\ e \mid \textsf{let}\ \overrightarrow{d}\ \textsf{in}\ e\ \textsf{end}$

Pattern

$p ::= c\ \overrightarrow{a}\ p \mid (p, p) \mid () \mid x$

Declaration

$d ::= \textsf{val}\ p = e \mid \textsf{idx}\ a = i \mid \textsf{type}\ \alpha = \tau \mid \textsf{absidx}\ a : s = i\ \textsf{with}\ \overrightarrow{d}\ \textsf{end}$

Figure 4-1: Syntax of Surface-TiML

4.1.1 Surface-TiML

The syntax of Surface-TiML is shown in Figure 4-1. Surface-TiML shares the same indices and sorts with μTiML. For types, Surface-TiML replaces the more primitive recursive types, existential types, and sum types with a single feature: datatypes. Each datatype consists of several constructors[5], each of which can construct a value of the datatype. Constructor names are denoted as c. Let us zoom in on the syntax for a datatype in Figure 4-1. Each datatype needs a name α to refer to itself in its constructors. A datatype can be parametrized on some type variables $\overrightarrow{\beta}$ and some sorts $\overrightarrow{s}$. The type of each constructor is a function that returns a value of type $\alpha\ \overrightarrow{\beta}\ \overrightarrow{i}$ given an input of some type τ. τ can be any type that is well-formed under the current context (e.g. τ can mention α and $\overrightarrow{\beta}$). The $\overrightarrow{i}$ in the constructor's result type can also be any indices as long as they are of sorts $\overrightarrow{s}$[6]. A constructor can introduce some local index variables $\overrightarrow{a}$ which $\overrightarrow{i}$ can mention[7]. As an example,

[5] Currently there must be at least one constructor for each datatype; datatypes with no constructors (representing the empty type) will be allowed in future versions.

[6] Note that the type arguments to the datatype in each constructor's result type are required to be $\overrightarrow{\beta}$ (no type-polymorphic recursion for datatypes). The reason for this is a technical constraint resulting from the translation of datatypes into μTiML's more primitive building blocks. Section 4.1.2 has a footnote explaining this technicality.

[7] Note that in τ and $\overrightarrow{i}$, the only available variables are α, $\overrightarrow{\beta}$, and $\overrightarrow{a}$. $\overrightarrow{s}$ do not introduce any variables.

below is the `list` datatype in Section 2.1 written in formal syntax:

$$\mathsf{list} \overset{\text{def}}{=} \mathsf{datatype}\ \alpha\ \beta\ \mathsf{Nat}\ \mathsf{as}\ [\ \ \mathsf{Nil} : \mathsf{unit} \to \alpha\ \beta\ 0,$$
$$\mathsf{Cons} : \forall n : \mathsf{Nat}.\ \beta \times \alpha\ \beta\ n \to \alpha\ \beta\ (n+1)\ ].$$

Note that α here is a self-reference name corresponding to the name `list` in Figure 2-1, while β here is the list-element type corresponding to *a* in Figure 2-1. I give the first constructor (`[]`) a name Nil here and the second constructor (`::`) a name Cons.

Surface-TiML's terms also have a large overlap with μTiML's. Surface-TiML has three term forms which μTiML does not have: pattern matching, constructor application, and compound let-binding. Pattern matching is the elimination form of datatypes, whose usage has been demonstrated by the code examples in Chapter 2. Note that the constructor pattern $c\ \overrightarrow{a}\ p$ can give names $\overrightarrow{a}$ to the constructor-local index variables (i.e. the $\overrightarrow{a : s}$ in the constructor's definition $c : \forall \overrightarrow{a : s}.\ \tau \to \alpha\ \overrightarrow{\beta}\ \overrightarrow{i}$). Constructor application is the introduction form of datatypes, constructing a value of a datatype using one of its constructors. Compound let-binding extends the simple let-binding in μTiML by allowing multiple bindings between let and in[8]. Each of these bindings is called a "declaration." Binding a value to a term variable is just one form of declaration, written as val $x = e$. The variable x can be generalized to be any pattern, which is the same as doing a pattern matching with just one branch. Other declaration forms include index/type definition (or "aliasing") and abstract-index definition. Abstract indices have been described in Chapter 2, whose special feature is that the definition of a can only be seen in the inner declarations $\overrightarrow{d}$, outside of which only the sort of a is visible.

Typing rules for Surface-TiML

Surface-TiML has its own type system, which again has a large overlap with μTiML's. I show in Figure 4-2 the kinding rule for datatype and the typing rule for absidx to help clarify the scoping of variables.

The kinding rule for datatype mainly checks that each constructor is well-kinded

[8]This feature is borrowed from Standard ML.

$\boxed{\Omega, \Lambda \vdash \tau :: \kappa}$, $\boxed{\Delta \vdash d : \Delta' \triangleright i}$ and $\boxed{\Delta \vdash e : \tau \triangleright i}$

$$\frac{\kappa \stackrel{\text{def}}{=} *^m \Rightarrow \overrightarrow{s} \Rightarrow * \qquad \begin{array}{c}\Omega \vdash \mathsf{wf}\ \overrightarrow{s} \\ \Omega, \Lambda; \alpha :: \kappa; \overrightarrow{\beta :: *} \vdash \forall \overrightarrow{a : s'}.\ \tau \xrightarrow{0} \alpha\ \overrightarrow{\beta}\ \overrightarrow{i} :: * \text{ (for each constructor)}\end{array}}{\Omega, \Lambda \vdash \mathsf{datatype}\ \alpha\ \beta_1 \cdots \beta_m\ \overrightarrow{s}\ \mathsf{as}\ \overrightarrow{c : \forall \overrightarrow{a : s'}.\ \tau \to \alpha\ \overrightarrow{\beta}\ \overrightarrow{i}} :: \kappa}\ \text{DATATYPE}$$

$$\frac{\Delta \vdash \mathsf{wf}\ s \qquad \Delta \vdash i : s \qquad \Delta; a : \{a : s \mid a = i\} \vdash \overrightarrow{d} : \Delta' \triangleright j}{\Delta \vdash \mathsf{absidx}\ a : s = i\ \mathsf{with}\ \overrightarrow{d}\ \mathsf{end} : (a : s; \Delta') \triangleright j}\ \text{ABSIDX}$$

$$\frac{\Delta \vdash \overrightarrow{d} : \Delta' \triangleright i_1 \qquad \Delta; \Delta' \vdash e : \tau \triangleright i_2 \qquad \tau,\ i_1 \text{ and } i_2 \text{ do not contain variables in } \Delta'}{\Delta \vdash \mathsf{let}\ \overrightarrow{d}\ \mathsf{in}\ e\ \mathsf{end} : \tau \triangleright i_1 + i_2}\ \text{LET}$$

$$\frac{\Delta(c) = (\tau_1, n, m) \qquad \Delta \vdash e : _ \triangleright j \qquad \Delta; x : \tau_1 \vdash x\ \overrightarrow{\tau}\ \overrightarrow{i}\ e : \tau_2 \triangleright _}{\Delta \vdash c\ \overrightarrow{\tau}\ \overrightarrow{i}\ e : \tau_2 \triangleright j + C_{\mathsf{Constr}}(|\overrightarrow{i}|, n, m)}\ \text{APPC}$$

Figure 4-2: Surface TiML kinding, typing, and declaration-checking rules (selected)

if seen as an arrow type $\forall \overrightarrow{a : s'}.\ \tau \xrightarrow{0} \alpha\ \overrightarrow{\beta}\ \overrightarrow{i}$. In other words, the kind checking for datatype delegates the job to that for arrow types. The cost of each constructor is determined by its definition (see rule APPC), not given via annotations, so a cost of zero is used as a placeholder in the arrow type. The kind of the whole datatype is $*^m \Rightarrow \overrightarrow{s} \Rightarrow *$, which is a shorthand for $*_1 \Rightarrow \cdots \Rightarrow *_m \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_n \Rightarrow *$.

Rule ABSIDX is one of the declaration-checking rules, which have the judgment form $\Delta \vdash d : \Delta' \triangleright i$. Δ' stands for variables newly introduced by the declaration d, and i is the cost of evaluating the declaration (e.g. a val declaration). A derived judgment $\Delta \vdash \overrightarrow{d} : \Delta' \triangleright i$ is for a sequence of declarations, in which case Δ' and i are the accumulated new-variable context and cost. Declaration checking is used to typecheck let as shown by rule LET, where there is a "no-escape" condition requiring that the typechecking results τ and $i_1 + i_2$ do not contain local variables visible only within the let-binding. In rule ABSIDX, the crucial arrangement is that when checking the inner declarations $\overrightarrow{d}$, index variables a's sort is $\{a : s \mid a = i\}$, while when exporting the new variable a to the outside world, its sort is changed to s.

I also show the typing rule for constructor applications. $\Delta(c) = (\tau_1, n, m)$ means looking up the information of constructor c in the context[9]. τ_1 is the type of the

[9]Datatype definitions are stored in the kinding context.

constructor, expressed as an arrow type (the type $\forall\overrightarrow{a:s'}.\ \tau \xrightarrow{0} \alpha\ \overrightarrow{\beta}\ \overrightarrow{i}$ in rule Datatype). n is the number of constructors that c's datatype has, and m is the position of c among its siblings (starting from 0). n and m are used for calculating the cost, which will be described in Section 4.5. Rule AppC delegates the job of typechecking $c\ \overrightarrow{\tau}\ \overrightarrow{i}\ e$ to typechecking $x\ \overrightarrow{\tau}\ \overrightarrow{i}\ e$ where x is of type τ_1[10].

4.1.2 Translating into μTiML

Translating datatypes

The translation of datatypes is shown below. The rule uses the translation of constructors which is shown below it. A datatype is translated into a higher-order recursive type, whose kind is $*^m \Rightarrow \overrightarrow{s} \Rightarrow *$. The body of the recursive type is a type-level function that takes in type arguments $\overrightarrow{\beta :: *}$ and index arguments $\overrightarrow{b:s}$ and returns a sum type. I use $\Sigma\overrightarrow{\tau}$ to mean the sum of a list of types, with right associativity[11]. Each of the components of the sum type is the translation of a constructor.

$$\begin{aligned}
&\left[\!\left[\mathsf{datatype}\ \alpha\ \beta_1 \cdots \beta_m\ s_1 \cdots s_n\ \mathsf{as}\ \overrightarrow{c : \forall a_1 : s'_1, \cdots, a_k : s'_k.\ \tau \to \alpha\ \overrightarrow{\beta}\ i_1 \cdots i_n}\right]\!\right] \\
&\qquad = \mu\alpha :: *^m \Rightarrow \overrightarrow{s} \Rightarrow *.\ \lambda\overrightarrow{\beta},\ b_1 : s_1, \cdots, b_n : s_n.\ \Sigma\overrightarrow{\left[\!\left[c : \forall\overrightarrow{a:s'}.\ \tau \to \alpha\ \overrightarrow{\beta}\ \overrightarrow{i}\right]\!\right]} \\
&\left[\!\left[c : \forall a_1 : s'_1, \cdots, a_k : s'_k.\ \tau \to \alpha\ \overrightarrow{\beta}\ i_1 \cdots i_n\right]\!\right] \\
&\qquad = \exists a_1 : s'_1, \cdots, a_k : s'_k,\ _ : \{b_1 = i_1 \wedge \cdots \wedge b_n = i_n\}.\ [\![\tau]\!]
\end{aligned} \tag{4.1.2.1}$$

In the constructor translation, the local index variables $\overrightarrow{a:s'}$ are translated to be existentially quantified variables. There is another existentially quantified index variable, $_ : \{b_1 = i_1 \wedge \cdots \wedge b_n = i_n\}$, whose name does not matter and whose sort[12] contains the information that $b_1 = i_1 \wedge \cdots \wedge b_n = i_n$[13]. The index variables $\overrightarrow{b}$ represent the index arguments of the constructed datatype, and $b_1 = i_1 \wedge \cdots \wedge b_n = i_n$ connects

[10] The cost estimation of $x\ \overrightarrow{\tau}\ \overrightarrow{i}\ e$ is not used, though the cost of e (j in rule AppC) is needed. In the typechecker implementation, j is returned after typechecking $x\ \overrightarrow{\tau}\ \overrightarrow{i}\ e$.

[11] $\overrightarrow{\tau}$ is required to be nonempty, so $\Sigma\overrightarrow{\tau}$ is well-defined.

[12] Remember that sort $\{\theta\}$ is a shorthand for the subset sort $\{_ : \mathsf{Unit}|\theta\}$.

[13] The encoding of polymorphic datatypes using equality constraints is inspired by [80].

the formal arguments $\overrightarrow{b}$ with the actual arguments $\overrightarrow{i}$ in each constructor.[14]

As an example, the translation result of the `list` datatype is shown below.

$$\mu\alpha :: * \Rightarrow \mathsf{Nat} \Rightarrow *.\ \lambda\beta :: *.\ \lambda b : \mathsf{Nat}.\ (\exists_ : \{b = 0\}.\ \mathsf{unit}) +$$
$$(\exists n : \mathsf{Nat},\ _ : \{b = n+1\}.\ \beta \times \alpha\ \beta\ n)$$

Translating constructor applications

$$[\![c_\pi\ \tau_1 \cdots \tau_m\ j_1 \cdots j_k\ e]\!] = \mathsf{fold}_{[\![\pi]\!]\ \overrightarrow{\tau}\ \overrightarrow{i}[\overrightarrow{j}/\overrightarrow{a}]}\ (\mathsf{inj}^{\#c,\#\pi}\ (\mathsf{pack}\ \langle \overrightarrow{j}; (), [\![e]\!]\rangle)) \tag{4.1.2.2}$$

The translation of constructor applications is shown above. The constructor c is assumed to be one of the constructors of the datatype in Equation 4.1.2.1, denoted as π. $\overrightarrow{\tau}$ and $\overrightarrow{j}$ are the actual type and index arguments; $\overrightarrow{\tau}$ will substitute for the formal type arguments $\overrightarrow{\beta}$, and $\overrightarrow{j}$ will substitute for the local index variables $\overrightarrow{a}$[15]. A constructor application is translated into a packing followed by an injection followed by a folding[16]. The type of the final result of this sequence of operations, as annotated in fold, is $[\![\pi]\!]\ \overrightarrow{\tau}\ \overrightarrow{i}[\overrightarrow{j}/\overrightarrow{a}]$. We can see that the actual arguments that will substitute for $\overrightarrow{b}$ are $\overrightarrow{i}[\overrightarrow{j}/\overrightarrow{a}]$, meaning $\overrightarrow{i}$ with variables $\overrightarrow{a}$ replaced by $\overrightarrow{j}$ ($\overrightarrow{a}$ and $\overrightarrow{j}$ are of equal lengths). In μTiML, fold, inj, and pack all require type annotations to guide typechecking; here I only show the annotation for fold, from which the other annotations are easy to figure out.

In the injection, I write $\#c$ to mean the position of the constructor c among its siblings (starting from 0) and $\#\pi$ to mean the number of π's constructors. $\mathsf{inj}^{p,n}$ is

[14]The reason for fixing the type arguments in the result type of each constructor to be $\overrightarrow{\beta}$ is that the Surface-TiML-to-μTiML translation uses an encoding where the type and index arguments to α in the result type are introduced as variables (called "result variables") . For indices, I can say that the local index variables are existentially quantified and are in a relation with the result variables. But for types, I cannot use this trick because type unification cannot depend on an arbitrary premise context as in index unification. When I fix the result type variables to be $\overrightarrow{\beta}$, each constructor's argument type is naturally expressed using $\overrightarrow{\beta}$, so I avoid this difficulty.

[15]Note that $\overrightarrow{j}$ will not substitute for the formal index arguments $\overrightarrow{b}$, as indicated by the length k instead of n.

[16]The sequence of these introduction forms corresponds to the sequence of types used in the translation of datatypes.

defined as[17]

$$\begin{array}{rcl} \mathsf{inj}^{0,1}\ e & = & e \\ \mathsf{inj}^{0,n}\ e & = & \mathsf{l}.e \\ \mathsf{inj}^{p,n}\ e & = & \mathsf{r}.(\mathsf{inj}^{p-1,n-1}\ e). \end{array} \tag{4.1.2.3}$$

The packing step packs indices $\overrightarrow{j}$ and the unit index () with $[\![e]\!]$[18]. The unit index is an instance for the sort $\{b_1 = i_1 \wedge \cdots \wedge b_n = i_n\}$. The condition $b_1 = i_1 \wedge \cdots \wedge b_n = i_n$ will be checked by the VC checker, according to the sorting rule for refinement sorts (see Figure 2-9). Note that this condition is trivially true because $\overrightarrow{b}$ have been replaced by $\overrightarrow{i}[\overrightarrow{j}/\overrightarrow{a}]$ when the typechecking reaches this point.

As an example, below is the translation of an expression involving applications of the two constructors from datatype `list`.

$$[\![\mathsf{Cons}\ \mathsf{int}\ 0\ (10, \mathsf{Nil}\ \mathsf{int}\ ())]\!] =$$
$$\mathsf{fold}_{[\![\mathsf{list}]\!]\ \mathsf{int}\ 1}\ (\mathsf{inj}^{1,2}\ (\mathsf{pack}\ \langle [0, ()], \mathsf{fold}_{[\![\mathsf{list}]\!]\ \mathsf{int}\ 0}\ (\mathsf{inj}^{0,2}\ (\mathsf{pack}\ \langle [()], ()\rangle))\rangle))$$

Translating pattern matchings

The section describes how to translate a compound pattern matching $\mathsf{case}\ e\ \mathsf{of}\ \overrightarrow{p \Rightarrow e}$ into a sequence of more primitive operations such as simple let-binding, pair projection, simple case-analysis for binary sum types, unfolding, and unpacking[19]. The pattern language used in compound pattern matching, shown in Figure 4-1, includes one that matches a datatype constructor, one that matches a pair, one that matches a unit value, and one that matches anything and gives it a name (or "alias"). In this section I will ignore the naming aspect and write the last pattern as _ (a wildcard pattern)[20]. I use pairs instead of tuples in this section to simplify the illustration. The implementation uses tuples. The input pattern-matching expression is assumed to have passed the Surface-TiML typechecker, which checks that the branches are

[17] inj is essentially a unary encoding of natural number p using the left and right injection as zero and successor. In the future I may change it to a binary encoding.

[18] I write pack with a list of indices as a shorthand for a series of single-index packs.

[19] The translation is similar to [60] though I was not aware of the work beforehand.

[20] Pattern aliases are used by the compiler as hints to choose variable names.

exhaustive and that there are no useless branches[21].

As a preparation step, each constructor pattern $c\ \overrightarrow{a}\ p$ is transformed into a combination of patterns comprised by three new pattern forms: $\mathsf{fold}\ p$ for matching a value of a recursive type, $\mathsf{pack}\ a\ p$ for matching a value of an (index-)existential type, and $\mathsf{inj}^{j,k}\ p$ for matching a value of a k-component sum type constructed by the j-th injector (starting from 0). The translation of a constructor pattern into these patterns is the same as the translation of a constructor application in Equation 4.1.2.2, except that $\overrightarrow{j}\,;()$ in pack is replaced with $\overrightarrow{a}\,;b$ where b is any name (not used) and type annotations are not needed.

The main complication in translating pattern matchings is the handling of pair patterns. When translating a pattern matching against a pair, the analysis of its second component needs to be postponed until the pattern matching for its first component is fully translated. So the input to the translation function includes the current pattern matching and all the postponed ones, which are in some sense "continuations" after the current pattern matching. I present the input to the translation function as a matrix:

$$\left[\!\!\left[\begin{matrix} e_1 & e_2 & \cdots & e_n & \\ p_{1,1} & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} & e'_m \end{matrix}\right]\!\!\right].$$

e_1 is the current matching target; $p_{1,1}$ to $p_{m,1}$ are the patterns in each branch (m branches in total); and e'_1 to e'_m are the branch expressions. The submatrix from e_2 to $p_{m,n}$ is the "continuation" part. To begin with, the translation of a pattern-

[21] The algorithm is adopted from [42]

matching expression[22] below is written as and equivalent to the right-hand side:

$$\left[\!\!\left[\begin{array}{l} \textsf{case}\ e\ \textsf{of} \\ \mid p_1 \Rightarrow e'_1 \\ \mid p_2 \Rightarrow e'_2 \\ \vdots \\ \mid p_m \Rightarrow e'_m \end{array}\right]\!\!\right] \equiv \left[\!\!\left[\begin{array}{ll} e & \\ p_1 & e'_1 \\ p_2 & e'_2 \\ \vdots & \vdots \\ p_m & e'_m \end{array}\right]\!\!\right].$$

At the beginning, there are no continuations. Continuations are introduced when matching pair patterns, as defined below.

$$\left[\!\!\left[\begin{array}{ccccc} e_1 & e_2 & \cdots & e_n & \\ (p_{1,1a}, p_{1,1b}) & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ (p_{2,1a}, p_{2,1b}) & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (p_{m,1a}, p_{m,1b}) & p_{m,2} & \cdots & p_{m,n} & e'_m \end{array}\right]\!\!\right] = \begin{array}{l} \textsf{let}\ x = [\![e_1]\!]\ \textsf{in} \\ \textsf{let}\ y_1 = x.1\ \textsf{in} \\ \textsf{let}\ y_2 = x.2\ \textsf{in} \\ \left[\!\!\left[\begin{array}{cccccc} y_1 & y_2 & e_2 & \cdots & e_n & \\ p_{1,1a} & p_{1,1b} & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ p_{2,1a} & p_{2,1b} & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{m,1a} & p_{m,1b} & p_{m,2} & \cdots & p_{m,n} & e'_m \end{array}\right]\!\!\right] \end{array}$$

To be a valid pattern matching, the patterns in all branches must be compatible with each other. In the case of pairs, all current patterns (those in the first column) must be pair patterns. A word on the wildcard pattern _ here[23]: if a pattern is required to be in a particular form but it is a wildcard, it is seen as a pattern in that form with all inner patterns being wildcards. For example, if a pattern is required to be a pair pattern but is actually _, it is seen as (_, _); if it is required to be a fold pattern, it is seen as fold _. Special care for wildcards is taken when translating inj patterns as described later. Note that each column of patterns can only have at most one required form; if there are two patterns in the same column with incompatible forms (e.g. a pair pattern and a fold pattern), the translation will fail (this situation

[22] I write a bar in each branch as an indicator for the start of that branch.

[23] Variable patterns have been converted into wildcard patterns.

is ruled out by the Surface-TiML typechecker).

In the translation for pair patterns above, the first-component patterns $p_{1,1a}$ to $p_{m,1a}$ become "current patterns" (first-column patterns) matching against the first component y_1 of expression e_1; the second component y_2 and its patterns $p_{1,1b}$ to $p_{m,1b}$ are pushed into the continuation to be dealt with later. The recursive call of the translation function is well-founded because the total size of the types of the "discriminees" (the match targets e_1 to e_n)[24] is decreased, because we replaced a product type with its two components (thus removed the product combinator). y_1 and y_2 are bound to the first and second projections of e_1. I use an intermediate variable x to make sure that e_1 is only evaluated once (since it may have side effects).

When all current patterns are wildcards or unit patterns, the translation for the current pattern matching is done and it moves on to the continuation part, as shown below.

$$\left[\!\!\left[\begin{matrix} e_1 & e_2 & \cdots & e_n & \\ _ & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ _ & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ _ & p_{m,2} & \cdots & p_{m,n} & e'_m \end{matrix}\right]\!\!\right] = \begin{array}{l} \mathsf{let}\ _ = [\![e_1]\!]\ \mathsf{in} \\ \left[\!\!\left[\begin{matrix} e_2 & \cdots & e_n & \\ p_{1,2} & \cdots & p_{1,n} & e'_1 \\ p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots \\ p_{m,2} & \cdots & p_{m,n} & e'_m \end{matrix}\right]\!\!\right] \end{array}$$

$$\left[\!\!\left[\begin{matrix} e_1 & e_2 & \cdots & e_n & \\ () & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ () & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ () & p_{m,2} & \cdots & p_{m,n} & e'_m \end{matrix}\right]\!\!\right] = \begin{array}{l} \mathsf{let}\ _ = [\![e_1]\!]\ \mathsf{in} \\ \left[\!\!\left[\begin{matrix} e_2 & \cdots & e_n & \\ p_{1,2} & \cdots & p_{1,n} & e'_1 \\ p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots \\ p_{m,2} & \cdots & p_{m,n} & e'_m \end{matrix}\right]\!\!\right] \end{array}$$

The translated expression starts with a let-binding of the match target e_1, because though the value of e_1 is not used, it still needs to be evaluated to trigger its side effects.

[24] Or equivalently the types of the patterns

The handling of fold and pack patterns is largely the same as that of pair patterns. In each case, it makes sure that the first-column patterns have the same form (wildcards are converted into the required form as discussed before) and then moves its focus onto the inner patterns. The respective elimination forms unfold and unpack are used to open up the discriminee expression and expose the inner part. In the pack-pattern case, note that the introduced local index variable a can be α-renamed to have the same name in each branch, so that this common name a can be used for unpack[25].

$$\left[\!\left[\begin{matrix} e_1 & e_2 & \cdots & e_n & \\ \mathsf{fold}\ p_1 & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ \mathsf{fold}\ p_2 & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathsf{fold}\ p_m & p_{m,2} & \cdots & p_{m,n} & e'_m \end{matrix}\right]\!\right] = \begin{matrix}\mathsf{let}\ x = \mathsf{unfold}\ [\![e_1]\!]\ \mathsf{in} \\ \left[\!\left[\begin{matrix} x & e_2 & \cdots & e_n & \\ p_1 & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ p_2 & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_m & p_{m,2} & \cdots & p_{m,n} & e'_m \end{matrix}\right]\!\right]\end{matrix}$$

$$\left[\!\left[\begin{matrix} e_1 & e_2 & \cdots & e_n & \\ \mathsf{pack}\ a\ p_1 & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ \mathsf{pack}\ a\ p_2 & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathsf{pack}\ a\ p_m & p_{m,2} & \cdots & p_{m,n} & e'_m \end{matrix}\right]\!\right] = \begin{matrix}\mathsf{unpack}\ [\![e_1]\!]\ \mathsf{as}\ \langle a, x\rangle\ \mathsf{in} \\ \left[\!\left[\begin{matrix} x & e_2 & \cdots & e_n & \\ p_1 & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ p_2 & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_m & p_{m,2} & \cdots & p_{m,n} & e'_m \end{matrix}\right]\!\right]\end{matrix}$$

The handling of injection patterns, shown below, is a bit more complicated, because (1) the branches need to be grouped according to the injector and (2) a wildcard should belong to all the groups because it matches any injector. The grouping should preserve the order of the branches because earlier branches take precedence over later ones. I express these considerations using the row-filtering operation, which applies a partial function to each pattern row (any row except for the first row $(e_1, e_2, \cdots, e_n)$) and discards the rows where the partial function is not defined. The partial function

[25] This subtlety is handled automatically by de Bruijn indices.

used for the j-th injector is select_j, which accepts a row if its first pattern is $\mathsf{inj}^{j,k}\ p$ (and transforms it to p) or a wildcard.

$$\left[\!\left[\begin{array}{lllll} & e_1 & e_2 & \cdots & e_n \\ \mathsf{inj}^{j_1,k}\ p_1 & p_{1,2} & \cdots & p_{1,n} & e'_1 \\ \mathsf{inj}^{j_2,k}\ p_2 & p_{2,2} & \cdots & p_{2,n} & e'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathsf{inj}^{j_m,k}\ p_m & p_{m,2} & \cdots & p_{m,n} & e'_m \end{array}\right]\!\right] = \begin{array}{l} \mathsf{case}\ [\![e_1]\!]\ \mathsf{of} \\ \mid \mathsf{inj}^{0,k}\ x \Rightarrow \left[\!\left[\begin{array}{l} x \quad e_2 \quad \cdots \quad e_n \\ \text{<rows filtered by } \mathsf{select}_0\text{>} \end{array}\right]\!\right] \\ \vdots \\ \mid \mathsf{inj}^{k-1,k}\ x \Rightarrow \left[\!\left[\begin{array}{l} x \quad e_2 \quad \cdots \quad e_n \\ \text{<rows filtered by } \mathsf{select}_{k-1}\text{>} \end{array}\right]\!\right] \end{array}$$

$$\begin{array}{lcl} \mathsf{select}_j(\mathsf{inj}^{j,k}\ p_1,\quad p_2, \cdots, p_n, e) & = & (p_1, p_2, \cdots, p_n, e) \\ \mathsf{select}_j(_,\qquad p_2, \cdots, p_n, e) & = & (_, p_2, \cdots, p_n, e) \end{array}$$

The translation result uses a case analysis for multi-component sum types, which is defined using the simple case analysis for binary sum types. Its definition is easy to figure out given the encoding of injectors in Equation 4.1.2.3.

The terminating case of the translation function is shown below, which applies when all the patterns have been translated into let-bindings, projections, etc., and it just needs to translate the branch expression. There could still be multiple branches even after we have walked through a pattern path[26], because wildcards can provide branches for injectors that have been covered by earlier branches. For example, if the first branch matches $\mathsf{inj}^{0,2}$ and the second branch matches _, there will be two branches applicable for the $\mathsf{inj}^{0,2}$ case. In such cases, the first branch takes precedence. m will always be at least 1 because the Surface-TiML typechecker checks that pattern matchings are exhaustive.

$$\left[\!\left[\begin{array}{l} e'_1 \\ e'_2 \\ \vdots \\ e'_m \end{array}\right]\!\right] = [\![e'_1]\!]$$

[26] And even when the Surface-TiML typechecker has ruled out useless branches

As an example, below is the translation of the `foldl` function in Figure 2-1.

$$
\begin{aligned}
\mathsf{foldl} &\stackrel{\text{def}}{=} \Lambda\alpha\ \beta : *.\ \mathsf{rec}\ g.\ \ e \\
e &\stackrel{\text{def}}{=} \Lambda m\ n : \mathsf{Nat}.\ \lambda f : \alpha \times \beta \xrightarrow{m} \beta.\ \lambda y : \beta.\ \lambda l : \mathsf{list}\ \alpha\ n. \\
&\quad\ \mathsf{case\ unfold}\ l\ \mathsf{of} \\
&\qquad z.\ \mathsf{unpack}\ z\ \mathsf{as}\ \langle _, _ \rangle\ \mathsf{in}\ y \\
&\quad\ \mathsf{or}\ z.\ \mathsf{unpack}\ z\ \mathsf{as}\ \langle n', w \rangle\ \mathsf{in\ unpack}\ w\ \mathsf{as}\ \langle _, u \rangle\ \mathsf{in}\ g\ m\ n'\ f\ (f(u.1, y))\ u.2 \\
e &: \forall m\ n : \mathsf{Nat}.\ (\alpha \times \beta \xrightarrow{m} \beta) \xrightarrow{0} \beta \xrightarrow{0} \mathsf{list}\ \alpha\ n \xrightarrow{(m+4)\times n} \beta
\end{aligned}
$$

g is the self-reference name of the recursive function. case here is the simple case analysis in Figure 2-4 for binary sum types. Some simple let-bindings are inlined.

Translating declarations (especially abstract indices)

The fact that Surface-TiML allows multiple declarations in a let-binding is not essential; such a let-binding can be transformed into a series of single-declaration let-bindings, as the following example shows.

$$
\begin{array}{l}
\mathsf{let} \\
\quad \mathsf{val}\ x = e_1 \\
\quad \mathsf{idx}\ a = i \\
\quad \mathsf{type}\ \alpha = \tau \\
\quad \mathsf{absidx}\ b = j\ \mathsf{with}\ \overrightarrow{d}\ \mathsf{end} \\
\mathsf{in} \\
\quad e_2 \\
\mathsf{end}
\end{array}
\Rightarrow
\begin{array}{l}
\mathsf{letval}\ x = e_1\ \mathsf{in} \\
\mathsf{letidx}\ a = i\ \mathsf{in} \\
\mathsf{lettype}\ \alpha = \tau\ \mathsf{in} \\
\mathsf{letabsidx}\ b = j\ \mathsf{with}\ \overrightarrow{d}\ \mathsf{in} \\
e_2
\end{array}
$$

letval is then translated into μTiML's simple let-binding. letidx and lettype are eliminated by inlining the index/type's definition[27]. letabsidx is translated into a pack

[27] This inlining will increase the size of the AST. In the future I will add letidx and lettype into μTiML and change its typechecking to carry index/type definitions in the typechecking context.

followed by an $\textsf{unpack}$, as shown below.

$$\left[\!\!\left[\begin{array}{l}\textsf{letabsidx}\ a = i\ \textsf{with val}\ x = e_1\ \textsf{in} \\ e_2\end{array}\right]\!\!\right] = \begin{array}{l}\textsf{unpack} \\ \quad \textsf{pack}\ \langle i, [\![e_1]\!][i/a]\rangle \\ \textsf{as}\ \langle a, x\rangle\ \textsf{in} \\ [\![e_2]\!]\end{array}$$

The translated form makes sure that the definition of a is visible in e_1 but not in e_2. Currently the translation only works on $\textsf{absidx}$ with a single $\textsf{val}$ as its inner declaration. In the future, multiple $\textsf{val}$s could be translated together as a tuple.

4.2 CPS conversion

The CPS-conversion phase is designed following the scheme in [63]. There are two complications brought about by TiML's unique setting that are not present in [63]: costs and states. I will explain these two complications when I delve into the definition of the translation function.

4.2.1 Type translation

$$\begin{array}{lcl}
\left[\!\!\left[\langle\phi_1,\tau_1\rangle \xrightarrow{i} \langle\phi_2,\tau_2\rangle\right]\!\!\right] & \stackrel{\text{def}}{=} & \forall j^{\textsf{Time}\times\textsf{Nat}},\ \phi^{\textsf{State}}.\ \langle\phi_1\cup\phi, [\![\tau_1]\!] \times \langle\phi_2\cup\phi, [\![\tau_2]\!]\rangle \xrightarrow{j} \blacksquare\rangle \xrightarrow{i+j} \blacksquare \\
[\![\forall^i_{a:s}.\ \tau]\!] & \stackrel{\text{def}}{=} & \forall a^s.\ \forall j^{\textsf{Time}\times\textsf{Nat}},\ \phi^{\textsf{State}}.\ \langle\phi, \langle\phi, [\![\tau]\!]\rangle \xrightarrow{j} \blacksquare\rangle \xrightarrow{i+j} \blacksquare \\
[\![\alpha]\!] & \stackrel{\text{def}}{=} & \alpha \\
[\![\textsf{int}]\!] & \stackrel{\text{def}}{=} & \textsf{int} \\
[\![\tau_1 \times \tau_2]\!] & \stackrel{\text{def}}{=} & [\![\tau_1]\!] \times [\![\tau_2]\!]
\end{array}$$

Figure 4-3: CPS conversion for types

Like all transformations in this compiler, the CPS conversion translates both types and terms. The translation for types is shown in Figure 4-3. To de-emphasize minor details and save space, in this section I write sort and type annotations as superscripts.

The first rule in Figure 4-3 conveys the essence of CPS conversion[28]: transforming a function that takes in an argument and returns a result, to a function that takes in an argument and a "continuation function" that accepts the result, and never returns. If we ignore all the cost and state aspects, the first rule can be simplified as

$$[\![\tau_1 \rightarrow \tau_2]\!] \stackrel{\mathsf{def}}{=} [\![\tau_1]\!] \times ([\![\tau_2]\!] \rightarrow \blacksquare) \rightarrow \blacksquare.$$

TiML does not allow functions that never return, so I choose to give bounds to the two arrows in the translated type[29]. The bound for the second arrow (i.e. the cost of the entire function) usually depends on the bound for the first arrow (the cost of the continuation). The cost of the continuation is universally quantified because the function should be able to work with any compatible continuation. In this section I follow the convention from Chapter 3 that a cost written as a single number could actually mean a cost pair (time and space). I write $\forall j^{\mathsf{Time} \times \mathsf{Nat}}$ to mean $\forall j_1^{\mathsf{Time}}\ j_2^{\mathsf{Nat}}$, and j will stand for (j_1, j_2) in its scope. The first rule in Figure 4-3 says that the cost of the whole function is $i + j$, the sum of the costs of the function body and the continuation.

Now let us move on to the issue of states. A function's precondition ϕ_1 and postcondition ϕ_2 usually only mention the state names that are used by the function, ignoring other state names that are present in the environment but not touched by the function. The continuation, however, may touch state names ignored by the function. Therefore, the translated version of the function should mention in its precondition both state names used by the original function and those used by the continuation. And since the translated function should be able to work with any continuation, the set of state names for the continuation should be universally quantified. This is where the $\forall \phi^{\mathsf{State}}$ comes from, and also the motivation of adding the base sort State to the index language. An index of sort State can be constructed by giving a concrete string-to-index map $\{\overrightarrow{u : i}\}$ or by using the "map union" index operator to combine

[28]The $\cup$ operator will be explained in the next paragraph.

[29]In a non-terminating setting, $\rightarrow \blacksquare$ should be interpreted as a function that never returns; in TiML's setting, it can be seen as $\rightarrow \langle \cdot, \mathsf{unit} \rangle$.

two indices[30]. The benefit of treating a state specification as an index (instead of always requiring it to be a concrete map) is that it can be universally quantified.

The translated function has a precondition $\phi_1 \cup \phi$, the union of the precondition of the original function and the "frame" part ϕ which the continuation will use and the function just passes along. The new postcondition is $\phi_2 \cup \phi$[31].

The translation of a $\forall$ type is very similar to that of an arrow type, since an arrow is just a degenerate form of $\forall$ in a dependent type system like Coq's. Here I only show the case for index-$\forall$; the case for type-$\forall$ is the same when replacing $a : s$ with $\alpha :: \kappa$. Remember that like arrows, a $\forall$ abstraction is a suspended computation that costs resources when triggered, so its type $\forall^i_{a:s}.\ \tau$ has the cost specification i (actually a pair of indices). The translated term is very similar to the one above, except that $\forall$'s pre- and postcondition are always empty[32]. Note that in the translated term, the $\forall a^s$ polymorphism no longer has costs, because the entire variable-introduction part $\forall a^s,\ j^{\mathsf{Time}\times\mathsf{Nat}},\ \phi^{\mathsf{State}}$ will be treated as a whole from now on[33].

The translation on other types is just a congruence, meaning it just recursively traverses the type, preserving its structure. The cases for type variables, base types, and product types are given as examples.

4.2.2 Term translation

The translation function $[\![e]\!]^{\phi}(k, j_k) = e', i$ for terms (shown in Figure 4-4) has four inputs: the term to be translated (e), the continuation (k), the cost of the continuation j_k, and the "frame" state (ϕ) that needs to be passed along to the continuation. The

[30] If there is an overlap of the domains of two maps, the second map overshadows the first in the union result.

[31] Note that the translated function usually is only well-typed when ϕ_1 and ϕ are disjoint. Since all continuations except for the initial one are constructed by the translation, and the initial one (picked by the compiler) has an empty precondition, this condition always holds. The typechecker currently is not able to validate the disjointness, so the soundness of the type system needs to rely on the fact that the compiler will never generate terms with overlapping unions. In the future, disjointness could be checked by having "disjoint" predicates in propositions.

[32] Because the computation a $\forall$ abstraction does is always creating a value (e.g. another abstraction) without using any state names, according to the value restriction for polymorphism

[33] In other words, before CPS conversion, index/type application (or "instantiation") has runtime effects (e.g. creating a new closure); after CPS conversion, index/type application is only for typechecking purposes and does not have runtime effects.

$$
\begin{array}{lcl}
\left[\!\left[(\lambda x.\ e)^{\langle\phi_1,\tau_1\rangle \xrightarrow{i} \langle\phi_2,\tau_2\rangle}\right]\!\right]^{\phi}(k, j_k) & & \\
 & \stackrel{\mathsf{def}}{=} & k\ (\forall j^{\mathsf{Time}\times\mathsf{Nat}},\ \phi^{\mathsf{State}}.\ \lambda^{\phi_1\cup\phi}(x^{[\![\tau_1]\!]}, c^{\langle\phi_2\cup\phi,[\![\tau_2]\!]\rangle \xrightarrow{j} \blacksquare}). \\
 & & \quad [\![e^{\tau_2}]\!]^{\phi}(c, j + C_{\mathsf{CpsAbs}_1}(e))),\ j_k + C_{\mathsf{CpsAbs}_2}(e) \\
\left[\!\left[e_1^{\phi_1,\tau_1}\ e_2^{\phi_2,\tau_2}\right]\!\right]^{\phi}(k, j_k) & \stackrel{\mathsf{def}}{=} & [\![e_1^{\tau_1}]\!]^{\phi}(\lambda^{\phi_1\cup\phi}x_1^{[\![\tau_1]\!]}.\ e_2',\ i_2) \\
\text{where} & & (e_2', i_2) = [\![e_2^{\tau_2}]\!]^{\phi}(\lambda^{\phi_2\cup\phi}x_2^{[\![\tau_2]\!]}.\ x_1\ (j_k + C_{\mathsf{CpsApp}_1}(k))\ (x_2, k)), \\
 & & \qquad\qquad i + j_k + C_{\mathsf{CpsApp}_2}(k)) \\
 & & _ \xrightarrow{i} _ = \tau_1 \\
\left[\!\left[(\Lambda a : s.\ e)^{\forall^{i}_{a:s}.\ \tau}\right]\!\right]^{\phi}(k, j_k) & \stackrel{\mathsf{def}}{=} & k\ (\forall a^{s}.\ \forall j^{\mathsf{Time}\times\mathsf{Nat}},\ \phi^{\mathsf{State}}.\ \lambda^{\phi}c^{\langle\phi,[\![\tau]\!]\rangle \xrightarrow{j} \blacksquare}. \\
 & & \quad [\![e^{\tau}]\!]^{\phi}(c, j + C_{\mathsf{CpsAbsT}_1}(e))),\ j_k + C_{\mathsf{CpsAbsT}_2}(e) \\
[\![e^{\phi_1,\tau}\ i]\!]^{\phi}(k, j_k) & \stackrel{\mathsf{def}}{=} & [\![e^{\tau}]\!]^{\phi}(\lambda^{\phi_1\cup\phi}x^{[\![\tau]\!]}.\ x\ i\ (j_k + C_{\mathsf{CpsAppT}_1}(k))\ k,\ j[i/a] + j_k + C_{\mathsf{CpsAppT}_2}(k)) \\
\text{where} & & \forall^{j}_{a:s}.\ _ = \tau \\
[\![x]\!](k, j_k) & \stackrel{\mathsf{def}}{=} & k\ x,\ j_k \\
[\![c]\!](k, j_k) & \stackrel{\mathsf{def}}{=} & k\ c,\ j_k + C_{\mathsf{Const}} \\
[\![\mathsf{rec}_{\tau}\ x.\ e]\!]^{\phi}(k, j_k) & \stackrel{\mathsf{def}}{=} & k\ (\mathsf{rec}_{[\![\tau]\!]}\ x.\ e')\ , j_k + i \\
\text{where} & & (e', i) = [\![e^{\tau}]\!]^{\phi}(\mathsf{id}, 0) \\
\left[\!\left[(\mathsf{case}\ e^{\phi',\tau'}\ \mathsf{of}\ x.e_1\ \mathsf{or}\ x.e_2)^{\tau}\right]\!\right]^{\phi}(k, j_k) & & \\
 & \stackrel{\mathsf{def}}{=} & \left[\!\left[e^{\tau'}\right]\!\right]^{\phi}(\lambda^{\phi'\cup\phi}y^{[\![\tau']\!]}.\ \mathsf{let}\ x_k = k\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ x.e_1'\ \mathsf{or}\ x.e_2', \\
 & & \quad C_{\mathsf{CpsCase}_0}(k) + \max(i_1, i_2 + C_{\textsc{JumpDest}})) \\
\text{where} & & (e_1', i_1) = [\![e_1^{\tau}]\!]^{\phi}(x_k, j_k + C_{\mathsf{CpsCase}}(e_1, k)) \\
 & & (e_2', i_2) = [\![e_2^{\tau}]\!]^{\phi}(x_k, j_k + C_{\mathsf{CpsCase}}(e_2, k)) \\
[\![o_{\mathsf{ut}}\ e^{\phi_1,\tau}]\!]^{\phi}(k, j_k) & \stackrel{\mathsf{def}}{=} & [\![e^{\tau}]\!](\lambda^{\phi_1\cup\phi}x^{[\![\tau]\!]}.\ k\ (o_{\mathsf{ut}}\ x),\ j_k + C_{\mathsf{CpsUnOp}}(o_{\mathsf{ut}})) \\
\left[\!\left[e_1^{\phi_1,\tau_1}\ o_{\mathsf{bt}}\ e_2^{\phi_2,\tau_2}\right]\!\right]^{\phi}(k, j_k) & \stackrel{\mathsf{def}}{=} & [\![e_1^{\tau_1}]\!](\lambda^{\phi_1\cup\phi}x_1^{[\![\tau_1]\!]}.\ e_2', i_2) \\
\text{where} & & (e_2', i_2) = [\![e_2^{\tau_2}]\!]^{\phi}(\lambda^{\phi_2\cup\phi}x_2^{[\![\tau_2]\!]}.\ k\ (x_1\ o_{\mathsf{bt}}\ x_2),\ j_k + C_{\mathsf{CpsBinOp}}(o_{\mathsf{bt}}, \tau_1, \tau_2, \phi_2)) \\
[\![\mathsf{pack}\ \langle i, e^{\phi_1,\tau}\rangle]\!]^{\phi}(k, j_k) & \stackrel{\mathsf{def}}{=} & [\![e^{\tau}]\!](\lambda^{\phi_1\cup\phi}x^{[\![\tau]\!]}.\ k\ (\mathsf{pack}\ \langle i, x\rangle),\ j_k + C_{\mathsf{Pack}}) \\
\left[\!\left[\mathsf{unpack}\ e_1^{\phi_1,\tau_1}\ \mathsf{as}\ \langle a, x\rangle\ \mathsf{in}\ e_2^{\tau_2}\right]\!\right]^{\phi}(k, j_k) & & \\
 & \stackrel{\mathsf{def}}{=} & [\![e_1^{\tau_1}]\!](\lambda^{\phi_1\cup\phi}x_1^{[\![\tau_1]\!]}.\ \mathsf{unpack}\ x_1\ \mathsf{as}\ \langle a, x\rangle\ \mathsf{in}\ e_2',\ i_2 + C_{\mathsf{Unpack}}) \\
\text{where} & & (e_2', i_2) = [\![e_2^{\tau_2}]\!]^{\phi}(k, j_k) \\
\left[\!\left[\mathsf{let}\ x = e_1^{\phi_1,\tau_1}\ \mathsf{in}\ e_2^{\tau_2}\right]\!\right]^{\phi}(k, j_k) & & \\
 & \stackrel{\mathsf{def}}{=} & [\![e_1^{\tau_1}]\!](\lambda^{\phi_1\cup\phi}x_1^{[\![\tau_1]\!]}.\ e_2',\ i_2) \\
\text{where} & & (e_2', i_2) = [\![e_2^{\tau_2}]\!]^{\phi}(k, j_k) \\
[\![e^{\tau} \triangleright i]\!]^{\phi}(k, j_k) & \stackrel{\mathsf{def}}{=} & [\![e^{\tau}]\!]^{\phi}(k, j_k) \triangleright i + j_k,\ i + j_k
\end{array}
$$

Figure 4-4: CPS conversion for terms

necessity of the last two inputs can be seen from the fact that they appear in the translated terms[34]. The translation result has two parts: the translated term (e') and the cost of the translated term (i). The latter is needed because of recursive calls of the translation function. I omit unused (or passed to recursive calls unchanged) parameters when writing translation rules. The translation often needs to know the types of the source term or its components, which I write as superscripts. These annotations will be provided by a typechecking phase before the CPS conversion[35]. The translated code will be let-normalized after CPS conversion, so an expression such as $e\ (x, k)$ when k is not a variable will be normalized to $\mathsf{let}\ y = k\ \mathsf{in}\ e\ (x, y)$.

The translation rules for lambda abstraction and function application (as the introduction and elimination form of arrow types), again, convey the essence of CPS conversion. The former should be easy to understand when read together with the first rule in Figure 4-3. A function (i.e. lambda abstraction) in direct style is translated into a function in continuation-passing style, and the new function (as a value) is given to the outer continuation k[36]. Note that when translating the function body e, the input continuation is the newly introduced variable c[37]. Cost annotations in the translation result need to be adjusted carefully to guarantee the translated terms to be well-typed and to avoid any precision loss. These cost adjustments are derived from the cost models of the source and target language of CPS conversion[38]. The definitions of the cost adjustments in Figure 4-4 are listed in Appendix B.2, and I will describe the cost models in Section 4.5.

The translation of function applications, if we ignore the continuation costs, can

---

[34]The translated terms use them in the annotations, which are needed for typechecking.

[35]In this compiler there is a typechecking phase between any two compilation phases, which is for both sanity checking the previous phase and providing annotations for the next phase. The AST of the input term to each translation can be thought to be fully annotated, where the type of each subtree is available, i.e. it can be called a "derivation tree" instead of a "syntax tree." In practice, to minimize AST size, the typechecker has switches to fine-control what annotations will be added. Each compilation phase sets these switches for the typechecker to generate only the annotations it needs, and consumes them to avoid passing unnecessary data to downstream phases.

[36]Note that for all value forms (e.g. $\lambda x.\ e$, $\Lambda a.\ e$, x, and c), the translation result is $k\ (\cdots)$, i.e. giving the translated value to the outer continuation k.

[37]And the inputs for continuation cost and frame state are the newly introduced variables j and ϕ.

[38]The derivation of these cost adjustments for CPS conversion took a large amount of coding and debugging time.

be written as

$$\left[\!\left[e_1^{\phi_1,\tau_1}\ e_2^{\phi_2,\tau_2} \right]\!\right]^{\phi}(k, j_k) \stackrel{\mathsf{def}}{=} [\![e_1^{\tau_1}]\!]^{\phi}(\lambda^{\phi_1 \cup \phi} x_1^{[\![\tau_1]\!]}.\ [\![e_2^{\tau_2}]\!]^{\phi}(\lambda^{\phi_2 \cup \phi} x_2^{[\![\tau_2]\!]}.\ x_1\ (j_k{+}C_{\mathsf{CpsApp}_1}(k))\ (x_2, k))).$$

The translated term can be understood in two parts. The first part is

$$[\![e_1^{\tau_1}]\!]^{\phi}(\lambda^{\phi_1 \cup \phi} x_1^{[\![\tau_1]\!]}.\ [\![e_2^{\tau_2}]\!]^{\phi}(\lambda^{\phi_2 \cup \phi} x_2^{[\![\tau_2]\!]}.$$

which is for evaluating e_1 and e_2 to values. This part is the same for all binary term operations (as can be seen in the $e_1\ o_{\mathsf{bt}}\ e_2$ case several lines below). e_1 is annotated with a state specification ϕ_1, which is the state after evaluating e_1[39].

The second part is

$$x_1\ (j_k + C_{\mathsf{CpsApp}_1}(k))\ (x_2, k)$$

which is the core of the translated term. This is how one uses a function (x_1) in continuation-passing style: by passing in the actual argument (x_2) and the continuation (k). Before the argument, the continuation cost is also provided to the function. Shrewd readers may find that another index argument, the frame state, is missing. Here I rely on a feature of TiML's typechecker: when the frame-state argument is missing, it can automatically infer it by calculating the difference between the current state and the function's precondition[40].

The translations of $\Lambda a.\ e$ and $e\ i$ are similar to the rules above and should be easy to understand after the previous description. The translation of $\mathsf{rec}\ x.\ e$ is a bit special. Ignoring continuation costs, it can be written as

$$[\![\mathsf{rec}_{\tau}\ x.\ e]\!]^{\phi}(k, j_k) \stackrel{\mathsf{def}}{=} k\ (\mathsf{rec}_{[\![\tau]\!]}\ x.\ [\![e^{\tau}]\!]^{\phi}(\mathsf{id}, 0)).$$

[39] This annotation is also provided by the pre-phase typechecking.

[40] This calculation is possible because an index of sort State will always be a union of concrete maps and variables. The calculation fails when one index cannot be shown to be a sub-state of the other index by comparing map keys and variable names. This failure will not occur for the terms generated by the CPS conversion. I could also have omitted the continuation-cost argument and let the compiler infer it. The inference is for making the CPS conversion simpler, but from a TCB-minimization standpoint, it is better to have a typechecker that does not do such inference.

Here I am using the translation function in a different way than other cases, where the translation function is always used with a continuation involving k. Here I use the identity function id as the continuation, and id will be inlined in the result. This use relies on the facts that e will always be a value (according to the value restriction for fixpoints) and that the translation result of a value will always be $k\ (\cdots)$. One can specialize e to e.g. $\lambda y.\ e'$ and unfold id and the translation function to make sense of the result.

The translations of $\mathsf{case}\ e\ \mathsf{of}\ x.e_1\ \mathsf{or}\ x.e_2$, $\mathsf{unpack}\ e_1\ \mathsf{as}\ \langle a, x\rangle\ \mathsf{in}\ e_2$, and $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ all have the form of firstly evaluating the first term and then evaluating later term(s) with continuation k. There is a let-binding in case's translation to avoid duplicating k in the result. The translations of unary term operation $o_{\mathsf{ut}}\ e$ and binary term operation $e_1\ o_{\mathsf{bt}}\ e_2$ are similar to and simpler than that for function applications. pack is similar to unary operations. The last rule translates cost annotation i into $i + j_k$[41].

4.3 Closure conversion

Closure conversion is for explicitly implementing a function's captured free variables (variables that are defined outside the function but used by it) as a tuple that will be passed in as an argument, so that each function will no longer have free variables and can be lifted to the top level. This compiler's closure-conversion phase follows that of [63] without much modification. The only difference is that in addition to type variables there are also index variables, and that a recursive function is translated together with the index/type abstractions surrounding it.

Closure conversion is a congruence in most cases[42], for both types and terms. The only non-congruence translation rules are shown in Figure 4-5, where the first rule is for types and last two are for terms[43]. For closure conversion the differences between

[41] Cost annotations cannot be discarded because they may be necessary for later typechecking.

[42] Meaning it just recurses down the structure

[43] The last rule covers all function definitions. If a function is not recursive, it is trivially turned to a recursive function by adding a rec at the beginning.

$$
\begin{array}{lcl}
\left[\!\left[\forall \overrightarrow{\alpha^{\kappa}}.\ \langle \phi, \tau \rangle \xrightarrow{i} \blacksquare \right]\!\right] & \stackrel{\text{def}}{=} & \exists \beta^*.\ (\forall \overrightarrow{\alpha^{\kappa}}.\ \langle \phi, \beta \times [\![\tau]\!] \rangle \xrightarrow{i} \blacksquare) \times \beta \\
\\
[\![e_1\ \overrightarrow{\tau}\ e_2]\!] & \stackrel{\text{def}}{=} & \mathsf{unpack}\ [\![e_1]\!]\ \mathsf{as}\ \langle \beta, x \rangle\ \mathsf{in}\ x.1\ \overrightarrow{[\![\tau]\!]}\ (x.2, [\![e_2]\!]) \\
\left[\!\left[\forall \overrightarrow{\alpha_a^{\kappa_a}}.\ \mathsf{rec}\ x.\ \forall \overrightarrow{\alpha_b^{\kappa_b}}.\ \lambda^{\phi} z^{\tau}.\ e \right]\!\right] & \stackrel{\text{def}}{=} & \mathsf{pack}\ \langle \tau_{env}, (v_{code}\ \overrightarrow{\beta}, (y_1, \cdots, y_m)) \rangle \\
 & \text{where} & \overrightarrow{\beta^{\kappa_c}} \quad = \quad FITV(\text{<source>}) \\
 & & y_1^{\tau_1}, \cdots, y_m^{\tau_m} \\
 & & \qquad = \quad FV(\text{<source>}) \\
 & & \tau_{env} \quad = \quad \tau_1 \times \cdots \times \tau_m \\
\text{generating top-level binding } v_{code} & = & \mathsf{rec}\ z_{code}.\ \forall \overrightarrow{\beta^{\kappa_c}},\ \overrightarrow{\alpha_a^{\kappa_a}},\ \overrightarrow{\alpha_b^{\kappa_b}}.\ \lambda^{\phi}(z_{env}^{\tau_{env}}, z^{[\![\tau]\!]}). \\
 & & \quad \mathsf{let}\ x = \mathsf{pack}\ \langle \tau_{env}, (z_{code}\ \overrightarrow{\beta}\ \overrightarrow{\alpha_a}, z_{env}) \rangle\ \mathsf{in} \\
 & & \quad \mathsf{let}\ y_1 = z_{env}.1\ \mathsf{in} \\
 & & \quad \vdots \\
 & & \quad \mathsf{let}\ y_m = z_{env}.m\ \mathsf{in} \\
 & & \quad [\![e]\!]
\end{array}
$$

Figure 4-5: Closure conversion

indices and types are not important so in Figure 4-5 I write a type list such as $\overrightarrow{\alpha}$ to mean a list of types and indices mixed together. $FV(\text{<source>})$ means the free term variables of the translation input; $FITV(\text{<source>})$ means free index/type variables. Note that after the translation, every function definition (e.g. v_{code}) is a rec followed by some ∀ and then a λ. Since every function definition (indicated by the leading rec) after closure conversion is closed, it can be freely moved. I write "generating top-level binding" to mean that the function definition (along with a variable that represents it) is put into a list which will be combined with the program's main body using let-bindings (i.e. function definitions are moved to the top level)[44].

Readers are referred to [63] for more explanation of the translation.

4.4 Code generation

The code-generation phase is responsible for translating TiML programs (terms) into TiEVM programs. The translation is shown from Figure 4-6 to 4-14, for both types

[44]A closed function can still refer to function names defined before it, so the order of function definitions needs to be preserved when moving them to the top level.

$$
\begin{array}{rcl}
\left[\!\left[\langle \phi, \tau \rangle \xrightarrow{i} \blacksquare\right]\!\right] & \stackrel{\mathsf{def}}{=} & (\phi, \{r_{\mathsf{arg}} \mapsto [\![\tau]\!]\}, \cdot) \xrightarrow{i} \blacksquare \\
[\![\mathsf{array}\ n\ \tau\ i]\!] & \stackrel{\mathsf{def}}{=} & \mathsf{arrayPtr}\ n\ [\![\tau]\!]\ i\ 32 \\
[\![\mathsf{tuple}\ \overrightarrow{\tau}]\!] & \stackrel{\mathsf{def}}{=} & \mathsf{tuplePtr}\ \overrightarrow{[\![\tau]\!]}\ 0\ \mathsf{false} \\
[\![\mathsf{ptr}\ \tau']\!] & \stackrel{\mathsf{def}}{=} & \mathsf{tuplePtr}\ \overrightarrow{[\![\tau]\!]}\ 0\ \mathsf{true} \\
 & \text{where} & \overrightarrow{\tau} = \mathsf{flattenTuple}(\tau') \\
[\![\mathsf{map}\ \tau']\!] & \stackrel{\mathsf{def}}{=} & \mathsf{map}\ (\mathsf{tuple}\ \overrightarrow{[\![\tau]\!]}) \\
 & \text{where} & \overrightarrow{\tau} = \mathsf{flattenTuple}(\tau')
\end{array}
$$

Figure 4-6: Code generation for types

and terms. The translation for types, shown in Figure 4-6, converts TiML arrow types (in CPS-ed form) to TiEVM code-pointer types. According to the converted type, the input argument is passed in a designated register r_{arg} (currently $r_{\mathsf{arg}} = r_{\mathsf{FirstGeneralReg}} = r_3$), and the stack at the entrance of each function should be empty. It converts arrays and tuples to array pointers (with offset 32 for skipping the length field) and in-memory-tuple pointers. The last two rules are for translating state types ptr and map. They use the operation flattenTuple, defined below, for flattening a tuple of tuples into a large single-layer tuple[45].

$$
\begin{array}{lcl}
\mathsf{flattenTuple}(\mathsf{tuple}\ [\tau_1, \cdots, \tau_n]) & = & \mathsf{flattenTuple}(\tau_1); \cdots ; \mathsf{flattenTuple}(\tau_n) \quad (n > 1) \\
\mathsf{flattenTuple}(\tau) & = & [\tau]
\end{array}
$$

There are four translation functions for terms, $[\![e]\!]$, $\mathcal{B}[\![e]\!]$, $\mathcal{C}[\![e]\!]$, and $\mathcal{P}[\![e]\!]$. I will focus on the former two first. The reason for having these two different forms of term-translation functions is that although all computations are expressed as terms in TiML, they are implemented by different facilities in TiEVM. Some terms are implemented by lists of instructions[46] without involving jumps and basic blocks, while some are implemented by several cooperating basic blocks. Because each block starts with a specification about the index/type/stack/register context at its entry and the cost bound, the second translation function needs to carry more parameters than the first.

[45]Remember from Chapter 3 that in-memory tuples of tuples are implemented by pointers while in-storage tuples of tuples are implemented by flattening.

[46]Not an “instruction sequence” as defined in Figure 3-1, which must end with a jump

$$
\begin{array}{lcl}
\llbracket x \rrbracket^{\gamma} & \stackrel{\mathsf{def}}{=} & \textsc{Push}\ \overline{32 \times r} \\
 & & \textsc{MLoad} \\
 & \text{when} & \gamma(x) = r \\
\llbracket x \rrbracket^{\gamma} & \stackrel{\mathsf{def}}{=} & \textsc{Push}\ l \\
 & \text{when} & \gamma(x) = l \\
\llbracket c \rrbracket & \stackrel{\mathsf{def}}{=} & \textsc{Push}\ \llbracket c \rrbracket \\
\llbracket \mathsf{state}\ u \rrbracket_{E} & \stackrel{\mathsf{def}}{=} & \textsc{Push}\ (\mathsf{state}\ E(u)) \\
\llbracket e_1\ o_{\mathsf{bt}}\ e_2 \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e_1 \rrbracket \\
 & & \llbracket e_2 \rrbracket \\
 & & \llbracket o_{\mathsf{bt}} \rrbracket \\
\llbracket e.n \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{Push}\ \overline{32 \times n} \\
 & & \textsc{Add} \\
 & & \textsc{MLoad} \\
\llbracket \mathsf{len}\ e \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{Push}\ \overline{32} \\
 & & \textsc{Swap1} \\
 & & \textsc{Sub} \\
 & & \textsc{MLoad} \\
\llbracket \mathsf{clear}\ e \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{Push}\ \overline{0} \\
 & & \textsc{Swap1} \\
 & & \textsc{SStore} \\
 & & \textsc{Push}() \\
\llbracket \mathsf{vectorLen}\ e \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{SLoad} \\
\llbracket \mathsf{storageGet}\ e \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{SLoad} \\
\llbracket \mathsf{iCellGet}\ e \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{SLoad} \\
\llbracket \mathsf{ptrProj}\ e\ n \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{Push}\ \overline{n} \\
 & & \textsc{Add} \\
 & & \textsc{RestrictView} \\
\llbracket e\ \tau \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{AppT}_{\llbracket \tau \rrbracket} \\
\llbracket e\ i \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{AppI}_{i} \\
\left\llbracket \mathsf{pack}_{\tau_1}\ \langle \tau_2, e \rangle \right\rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{Pack}^{\llbracket \tau_1 \rrbracket}_{\llbracket \tau_2 \rrbracket} \\
\llbracket \mathsf{pack}_{\tau}\ \langle i, e \rangle \rrbracket & \stackrel{\mathsf{def}}{=} & \llbracket e \rrbracket \\
 & & \textsc{PackI}^{\llbracket \tau \rrbracket}_{i}
\end{array}
$$

Figure 4-7: Code generation for terms

$$
\begin{array}{lcl}
[\![\mathsf{fold}_\tau\ e]\!] & \stackrel{\text{def}}{=} & [\![e]\!] \\
 & & \textsc{Fold}_{[\![\tau]\!]} \\
[\![\mathsf{unfold}_\tau\ e]\!] & \stackrel{\text{def}}{=} & [\![e]\!] \\
 & & \textsc{Unfold}_{[\![\tau]\!]} \\
[\![e : \tau]\!] & \stackrel{\text{def}}{=} & [\![e]\!] \\
 & & \textsc{AscType}_{[\![\tau]\!]} \\
[\![\mathsf{never}_\tau]\!] & \stackrel{\text{def}}{=} & \textsc{Push}\ \mathsf{never}_{[\![\tau]\!]} \\
[\![\mathsf{l}_\tau.e]\!] & \stackrel{\text{def}}{=} & [\![e]\!] \\
 & & \textsc{Push}\ \overline{\mathsf{false}} \\
 & & \textsc{Inj}_{[\![\tau]\!]} \\
[\![(e_1^{\tau_1}, \cdots, e_n^{\tau_n})]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!] \\
 & & \vdots \\
 & & [\![e_n]\!] \\
 & & \textsc{TupleMalloc}_{\overrightarrow{[\![\tau]\!]}} \\
 & & \textsc{Push}\ \overline{32 \times n} \\
 & & \textsc{Add} \\
 & & \left\{ \begin{array}{l} \textsc{Push}\ \overline{32} \\ \textsc{Swap1} \\ \textsc{Sub} \\ \textsc{Swap1} \\ \textsc{TupleInit} \end{array} \right\} \text{repeat } n \text{ times} \\
 & & \textsc{TupleDone} \\
[\![\mathsf{arrayFromList}_m^\tau\ \{e_1^{\tau_1}, \cdots, e_n^{\tau_n}\}]\!] & \stackrel{\text{def}}{=} & \textsc{Push}\ \overline{n} \\
 & & \textsc{ArrayMalloc}_{[\![\tau]\!]}^{\mathsf{true}}\ m \\
 & & \textsc{Swap1} \\
 & & \textsc{ArrayInitLen} \\
 & & \textsc{Push}\ \overline{0} \\
 & & \left\{ \begin{array}{l} [\![e_i]\!] \\ \textsc{Swap2} \\ \textsc{Swap1} \\ \textsc{ArrayInit}_m \\ \textsc{Swap2} \\ \textsc{Pop} \\ \textsc{Swap1} \\ \textsc{Push}\ \overline{m} \\ \textsc{Add} \end{array} \right\} \text{for } i = 1 \text{ to } n \\
 & & \textsc{Pop} \\
 & & \textsc{ArrayDone}
\end{array}
$$

Figure 4-8: Code generation for terms (continued)

$$
\begin{array}{lll}
[\![\mathsf{read}_{32}\ e_1\ e_2]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & \textsc{Push}\ \overline{32} \\
 & & \textsc{Mul} \\
 & & \textsc{Add} \\
 & & \textsc{MLoad} \\
[\![\mathsf{read}_{1}\ e_1\ e_2]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & \textsc{Add} \\
 & & \textsc{Push}\ \overline{31} \\
 & & \textsc{Swap1} \\
 & & \textsc{Sub} \\
 & & \textsc{MLoad} \\
 & & \textsc{Int2Byte} \\
[\![\mathsf{write}_{32}\ e_1\ e_2\ e_3]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & [\![e_3]\!] \\
 & & \textsc{Swap2} \\
 & & \textsc{Swap1} \\
 & & \textsc{Push}\ \overline{32} \\
 & & \textsc{Mul} \\
 & & \textsc{Add} \\
 & & \textsc{MStore} \\
 & & \textsc{Push}\ () \\
[\![\mathsf{write}_{1}\ e_1\ e_2\ e_3]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & [\![e_3]\!] \\
 & & \textsc{Swap2} \\
 & & \textsc{Add} \\
 & & \textsc{MStore8} \\
 & & \textsc{Push}\ () \\
[\![\mathsf{mapPtr}\ e_1\ e_2]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & \textsc{MapPtr} \\
[\![\mathsf{storageSet}\ e_1\ e_2]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & \textsc{Swap1} \\
 & & \textsc{SStore} \\
 & & \textsc{Push}\ () \\
[\![\mathsf{iCellSet}\ e_1\ e_2]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & \textsc{Swap1} \\
 & & \textsc{SStore} \\
 & & \textsc{Push}\ ()
\end{array}
$$

Figure 4-9: Code generation for terms (continued)

$$
\begin{array}{lcl}
[\![\mathsf{vectorGet}\ e_1\ e_2]\!] & \stackrel{\mathsf{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & \textsc{Swap1} \\
 & & \textsc{VectorPtr} \\
 & & \textsc{SLoad} \\
[\![\mathsf{vectorSet}\ e_1\ e_2\ e_3]\!] & \stackrel{\mathsf{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & [\![e_3]\!] \\
 & & \textsc{Swap2} \\
 & & \textsc{VectorPtr} \\
 & & \textsc{SStore} \\
 & & \textsc{Push}\ () \\
[\![\mathsf{pushback}\ e_1\ e_2]\!] & \stackrel{\mathsf{def}}{=} & [\![e_1]\!] \\
 & & [\![e_2]\!] \\
 & & \textsc{VectorPushBack} \\
 & & \textsc{Push}\ ()
\end{array}
$$

Figure 4-10: Code generation for terms (continued)

Let us focus on the simplest translation function $[\![e]\!]$ first. When all the parameters are spelled out, its full form is $[\![e]\!]^{\gamma}_{E}$, where γ is a mapping from variable names to registers or labels. E is a mapping from state names to integers (as the encodings of the state names). As before, I omit unused (or passed to recursive calls unchanged) parameters when writing translation rules. Each generated instruction list does not consume anything on the stack and will put one word (the result of evaluating the expression) on top of the stack. It may use the stack to store intermediate results during the evaluation[47].

Most of the rules for $[\![e]\!]$ are straightforward. The translation of a variable x depends on whether $\gamma(x)$ is a register or a code label (i.e. a variable could be implemented by either a register or a code label). I write $\overline{n}$ as a value of an indexed natural-number type (instead of a value of type int), which is often used for Push[48]. The constant translation $[\![c]\!]$ is trivial and omitted here. ptrProj, the projection of an in-storage-tuple pointer, is translated into RestrictView, as discussed in Section

[47]Note that the translation allows compound expressions, so the input expressions are not required to be in let-normal form, though after previous compilation phases most of them are. Evaluating compound expressions may cause stack overflow (will not happen for this compiler because of CPS conversion), which will be detected by statically analyzing the stack depth in a future version.

[48]Push often needs its argument to be $\overline{n}$ instead of n because the value will influence typechecking.

3.4.7. $\mathsf{l}.e$, the left injection, is translated into INJ with a $\overline{\mathsf{false}}$ on top of the stack, as discussed in Section 3.4.2. A tuple is implemented as a TUPLEMALLOC followed by a series of TUPLEINITs followed by a TUPLEDONE. arrayFromList, for creating an array with a list of elements, is implemented in a similar fashion with ARRAYMALLOC, ARRAYINIT, and ARRAYDONE. Note the true parameter to ARRAYMALLOC to indicate that the array will be initialized upward (from element 0 to element $n-1$) and the m parameter for element width. For array reading and writing, the translation only supports element widths of 1 and 32 (in bytes) so far.

Now let us move on to the translation function $\mathcal{B}[\![e]\!]$, whose full form is $\mathcal{B}[\![e]\!]^{\gamma}_{\Lambda,R,E}$. γ and E have the same meanings as before. Λ stands for the combined index-and-type context of the input expression. R is the current register-typing context. These two are needed when generating new basic blocks. The output of this translation must be an instruction sequence, ending with a jump or halt. It can optionally generate additional blocks. Any unintroduced names in the result code (such as a register r or a label l) are assumed to be fresh[49]. Because the output must be an instruction sequence, the input must be a "continuation" expression, meaning it can be the body of a CPS-ed function. Such expressions include let-binding, unpacking, function application, branching, halt, and a continuation expression annotated with a cost[50].

The translation for $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ first translates e_1, then puts the evaluation result in a fresh register before translating e_2. Note that e_1 is translated by $[\![\ ]\!]$ while e_2 is translated by $\mathcal{B}[\![\ ]\!]$. unpack is translated similarly, except that the index variable a is added to Λ when translating e_2[51]. A function application is simply implemented by storing the argument to register r_{arg} and then jumping. Branching is one of those places where extra blocks need to be generated. The three forms of branching, if, ifi, and case, are all translated in a fashion where the first branch is implemented by the main output while the second branch is implemented as an extra code block[52]. I

[49]There is a register allocator and a label allocator that return ever-larger register and label numbers. The register allocator is reset for each top-level function.

[50]After CPS conversion, only continuation expressions may have cost annotations.

[51]unpack for type existentials is similar.

[52]JUMPDEST is a noop instruction that marks the destination of a jump. The EVM specifica-

$$\mathcal{B}[\![\text{let } x = e_1^{\tau} \text{ in } e_2]\!]^{\gamma}_{\Lambda,R,E} \stackrel{\text{def}}{=} \begin{array}{l} [\![e_1]\!] \\ \textsc{Push}\ \overline{32 \times r} \\ \textsc{MStore} \\ \mathcal{B}[\![e_2]\!]^{\gamma[x \mapsto r]}_{\Lambda,R[r \mapsto [\![\tau]\!]],E} \end{array}$$

$$\mathcal{B}\left[\!\!\left[\text{unpack } e_1^{\exists a:s.\tau} \text{ as } \langle a, x \rangle \text{ in } e_2\right]\!\!\right]^{\gamma}_{\Lambda,R,E} \stackrel{\text{def}}{=} \begin{array}{l} [\![e_1]\!] \\ \textsc{Push}\ \overline{32 \times r} \\ \textsc{MStore} \\ \mathcal{B}[\![e_2]\!]^{\gamma[x \mapsto r]}_{\Lambda;a:s,R[r \mapsto [\![\tau]\!],E]} \end{array}$$

$$\mathcal{B}[\![e_1\ e_2]\!] \stackrel{\text{def}}{=} \begin{array}{l} [\![e_1]\!] \\ [\![e_2]\!] \\ \textsc{Push}\ \overline{32 \times r_{\text{arg}}} \\ \textsc{MStore} \\ \textsc{Jump} \end{array}$$

$$\mathcal{B}\left[\!\!\left[\text{if } e^{\phi} \text{ then } e_1 \text{ else } (e_2 \triangleright i)\right]\!\!\right]^{\gamma}_{\Lambda,R,E} \stackrel{\text{def}}{=} \begin{array}{l} [\![e]\!]^{\gamma}_{E} \\ \textsc{IsZero} \\ \textsc{Push}\ (l\ \Lambda) \\ \textsc{JumpI} \\ \mathcal{B}[\![e_1]\!]^{\gamma}_{\Lambda,R,E} \end{array}$$

$$\text{generating block}\quad l \mapsto \text{block } \{ \begin{array}{l} \forall \Lambda.\ (\phi, R, \cdot) \xrightarrow{i + C_{\textsc{JumpDest}}} \blacksquare : \\ \textsc{JumpDest} \\ \mathcal{B}[\![e_2]\!]^{\gamma}_{\Lambda,R,E}\ \} \end{array}$$

$$\mathcal{B}\left[\!\!\left[\text{ifi } e^{\phi, \text{ibool } j} \text{ then } x.e_1 \text{ else } (x.e_2 \triangleright i)\right]\!\!\right]^{\gamma}_{\Lambda,R,E} \stackrel{\text{def}}{=} \begin{array}{l} [\![e]\!]^{\gamma}_{E} \\ \textsc{IsZero} \\ \textsc{Push}\ () \\ \textsc{Swap1} \\ \textsc{Push}\ (l\ \Lambda) \\ \textsc{JumpI} \\ \textsc{Push}\ \overline{32 \times r} \\ \textsc{MStore} \\ \mathcal{B}[\![e_1]\!]^{\gamma[x \mapsto r]}_{\Lambda,R[r \mapsto \exists\{j = \text{true}\}.\text{unit}],E} \end{array}$$

$$\text{generating block}\quad l \mapsto \text{block } \{ \begin{array}{l} \forall \Lambda.\ (\phi, R, [\tau_2]) \xrightarrow{i + C_{\textsc{JumpDest}}} \blacksquare : \\ \textsc{JumpDest} \\ \textsc{Push}\ \overline{32 \times r} \\ \textsc{MStore} \\ \mathcal{B}[\![e_2]\!]^{\gamma[x \mapsto r]}_{\Lambda,R[r \mapsto \tau_2],E}\ \} \end{array}$$

$$\text{where}\quad \tau_2 = \exists\{j = \text{false}\}.\text{unit}$$

Figure 4-11: Code generation for terms (outputing instruction sequence)

$$
\begin{array}{lcl}
\mathcal{B}[\![\textsf{case } e^{\phi,\tau_1+\tau_2} \textsf{ of } x.e_1 \textsf{ or } x.(e_2 \triangleright i)]\!]^{\gamma}_{\Lambda,R,E} & \overset{\textsf{def}}{=} & [\![e]\!]^{\gamma}_{E} \\
 & & \textsc{Push}\ (l\ \Lambda) \\
 & & \textsc{BrSum} \\
 & & \textsc{Push}\ \overline{32} \\
 & & \textsc{Add} \\
 & & \textsc{MLoad} \\
 & & \textsc{Push}\ \overline{32 \times r} \\
 & & \textsc{MStore} \\
 & & \mathcal{B}[\![e_1]\!]^{\gamma[x \mapsto r]}_{\Lambda,R[r \mapsto [\![\tau_1]\!]],E} \\
 & \text{generating block} & l \mapsto \textsf{block}\ \{ \\
 & & \forall\Lambda.\ (\phi', R, [\textsf{ibool true} \times [\![\tau_2]\!]]) \xrightarrow{i+C_{\textsc{JumpDest}}} \blacksquare : \\
 & & \textsc{JumpDest} \\
 & & \textsc{Push}\ \overline{32} \\
 & & \textsc{Add} \\
 & & \textsc{MLoad} \\
 & & \textsc{Push}\ \overline{32 \times r} \\
 & & \textsc{MStore} \\
 & & \mathcal{B}[\![e_2]\!]^{\gamma[x \mapsto r]}_{\Lambda,R[r \mapsto [\![\tau_2]\!]],E}\ \} \\
\mathcal{B}[\![\textsf{halt } e^{\tau}]\!] & \overset{\textsf{def}}{=} & [\![e]\!] \\
 & & \textsc{Halt}_{[\![\tau]\!]} \\
\mathcal{B}[\![e \triangleright i]\!] & \overset{\textsf{def}}{=} & \textsc{AscTime}_i \\
 & & \mathcal{B}[\![e]\!]
\end{array}
$$

Figure 4-12: Code generation for terms (outputing instruction sequence, continued)

$$\mathcal{B}\left[\!\!\left[\mathsf{let}\ x = \mathsf{new}_n\ e_1^{\mathsf{nat}\ i}\ e_2^{\phi,\tau}\ \mathsf{in}\ (e \triangleright (j_1, j_2))\right]\!\!\right]^{\gamma}_{\Lambda,R,E} \stackrel{\mathsf{def}}{=}$$

$[\![e_1]\!]^{\gamma}_{E}$
$[\![e_2]\!]^{\gamma}_{E}$
SWAP1
DUP1
ARRAYMALLOC$^{\mathsf{false}}_{[\![\tau]\!]}\ n$
DUP2
ARRAYINITLEN
SWAP1
PUSH $\overline{n}$
MUL
PUSH $(l_1\ \Lambda\ i)$
JUMP

generating blocks

$l_1 \mapsto \mathsf{block}\ \{$
$\forall \Lambda,\ a : \{a : \mathsf{Nat} \mid a <= i\}.$
$(\phi, R, [\mathsf{nat}\ (n \times a), \mathsf{preArray}\ [\![\tau]\!]\ i\ a\ \mathsf{true}\ \mathsf{false}, [\![\tau]\!]])$
$\xrightarrow{a \times C_{\mathsf{NewLoop}}(n) + C_{\mathsf{NewLoopTest}} + C_{\mathsf{NewPostLoop}} + j_1,\ j_2} \blacksquare :$
JUMPDEST
PUSH ()
DUP2
ISZERO
PUSH $(l_2\ \Lambda\ a)$
JUMPI
UNPACKI
ASCTIME$_{a \times C_{\mathsf{NewLoop}}(n) + C_{\mathsf{NewPostLoop}} + j_1}$
ASCSPACE$_{j_2}$
POP
PUSH $\overline{n}$
SWAP1
SUB
ARRAYINIT$_n$
PUSH $(l_1\ \Lambda\ (a-1))$
JUMP }

$l_2 \mapsto \mathsf{block}\ \{$
$\forall \Lambda,\ a : \mathsf{Nat}.$
$(\phi, R, [\exists \{a \times n = 0\}.\mathsf{unit}, \mathsf{nat}\ (n \times a),$
$\mathsf{preArray}\ [\![\tau]\!]\ i\ a\ \mathsf{true}\ \mathsf{false}, [\![\tau]\!]])$
$\xrightarrow{C_{\mathsf{NewPostLoop}} + j_1,\ j_2} \blacksquare :$
JUMPDEST
UNPACKI
POP
POP
SWAP1
POP
ARRAYDONE
PUSH $\overline{32 \times r}$
MSTORE
$\mathcal{B}[\![e]\!]^{\gamma[x \mapsto r]}_{\Lambda, R[r \mapsto \mathsf{arrayPtr}\ [\![\tau]\!]\ i\ 32], E}$ }

Figure 4-13: Code generation for terms (outputing instruction sequence, continued)

$$\mathcal{C}\left[\!\!\left[(\forall\Lambda.\ \lambda^{\phi}x^{\tau}.\ e)^{\tau'}\right]\!\!\right]^{\gamma}_{E} \stackrel{\mathsf{def}}{=} \begin{array}{l} \mathsf{block}\ \{\ \forall\Lambda.\ (\phi, R, \cdot) \xrightarrow{i} \blacksquare : \\ \quad \text{JUMPDEST} \\ \quad \mathcal{B}[\![e]\!]^{\gamma[x \mapsto r_{\mathsf{arg}}]}_{\Lambda,R,E}\ \} \end{array}$$

$$\text{where}\quad R = \{r_{\mathsf{arg}} \mapsto [\![\tau]\!]\}$$

$$\forall_.\ _ \xrightarrow{i} \blacksquare = \tau'$$

$$\mathcal{P}\left[\!\!\left[\begin{array}{l} \mathsf{let}\ x_1 = \mathsf{rec}_{\tau_1}\ x_1.\ e_1\ \mathsf{in} \\ \vdots \\ \mathsf{let}\ x_n = \mathsf{rec}_{\tau_n}\ x_n.\ e_n\ \mathsf{in} \\ e \end{array}\right]\!\!\right]^{\gamma}_{\Lambda,R,E} \stackrel{\mathsf{def}}{=} \mathcal{B}[\![e]\!]^{\gamma_n}_{\cdot,\cdot,E}$$

$$\text{where}\quad \gamma_i = \{x_1 \mapsto l_1, \cdots, x_i \mapsto l_i\}\ \text{for each}\ i$$

$$\text{generating block}\quad l_i \mapsto \mathcal{C}[\![e_i^{\tau_i}]\!]^{\gamma}_{E}\ \text{for each}\ i$$

Figure 4-14: Code generation for top-level functions and programs

write $l\ \Lambda$ to mean l applied to all the index and type variables in Λ, in order. Note that a cost annotation for the second branch is needed in order to generate the code block. In the translation of ifi and case, prelude code is needed for both branches to store the value on top of the stack to the register representing x.

The last rule for $\mathcal{B}[\![\ ]\!]$ is a special case of let-binding where the first expression is new $e_1\ e_2$. The implementation of new is essentially a mini for-loop: the main output is the code before the loop, block l_1 is the loop body, and block l_2 is the code after the loop. The specifications for blocks l_1 and l_2 need to be carefully tuned to be both sound (the specification is no less than the actual cost) and accurate (the specification is close to the actual cost). new must be translated together with its continuation (as a let-binding) because the translation of the continuation, $\mathcal{B}[\![e]\!]$, is needed to form block l_2.

The other translation functions, $\mathcal{C}[\![e]\!]$ and $\mathcal{P}[\![e]\!]$, are for translating a top-level function definition[53] into a basic block and translating the whole program. After closure conversion, the input TiML program is in the form of a series of top-level recursive function definitions (defined using let-bindings) followed by a main program

tion stipulates that any jump instruction must jump to an instruction boundary where the next instruction is JUMPDEST, in order to prevent malicious jumps into the middle of an instruction.

[53]After closure conversion, all function definitions are top-level.

body, as shown by the input to $\mathcal{P}[\![\ ]\!]$. These function definitions are closed (except for the function names defined before). After translating the whole program, generated basic blocks are collected and combined with the translated main body to form a TiEVM program.

4.5 Derived cost models

The only ground truth about costs for Ethereum smart contracts is the official cost model in the EVM specification. This directly determines the cost parameters in TiEVM. Going up from there, the cost models for higher-level intermediate and source languages need to be derived from lower ones, reflecting the compiler transformations performed between each two consecutive levels. Following this bottom-up direction, I will first describe the cost model for the TiML language used between closure conversion and code generation, and then describe the one between CPS conversion and closure conversion, and so on up to the one for the Surface-TiML language. Note that except for Surface-TiML, all the others are the same language (the μTiML language) but with different cost models.

4.5.1 TiEVM cost model

TiEVM's cost model consists of all the costs in the typing rules in Section 3.4. Most of them are parameters such as C_{MLoad} and C_{Push} that are defined in the EVM specification. Some cost parameters such as C_{MapPtr} are for pseudo-instructions that can be easily derived by expanding the pseudo-instructions.

4.5.2 TiML cost model before code generation

A TiML cost model consists of the definitions of the cost parameters in Figure 2-10. Most of the definitions do not differ across compilation stages. These definitions are listed in Appendix B.1. The ones that do are the costs for function application, abstraction, index/type application, index/type polymorphism, and branching. Before

delving into these interesting costs, I will first discuss variables and let-bindings.

The costs for variables and let-bindings are the same on all stages: zero. This is due to an accounting scheme I use: each operation is assumed to read its operands from variables and write its result to a variable by a let-binding, and the costs for variable reading/writing are paid by the operation, not the variable or the let-binding. The motivation is that the sequence of operations is relatively stable across compilation stages (the compiler is not supposed to freely insert new operations) while the structure of let-bindings can be radically different. For example, the source program can be a large compound expression without any let-binding, while the target program can be let-normalized so that each operation is followed by a let-binding. By adopting this accounting scheme, I am conservatively assuming that a maximal number of let-bindings will be inserted. I am also assuming that there are no let-bindings that are just variable renaming (because no operation is accounting for such costs). A simple inlining can remove all such useless let-bindings.

From Figure 2-10 we can see that the cost of a function application depends on the cost function $C_{\mathsf{App}}(n_k, b_k)$, which takes in two parameters n_k and b_k. n_k is the number of live term variables after this application, which equals to the number of free term variables in the "continuation." The continuation is the piece of program from after this application to the end of this function or branch. b_k is a Boolean indicating whether such a continuation exists (it does not exist e.g. when the application is at the end of a function, i.e. a tail call). These two parameters are calculated by a backward program analysis (a live-variable analysis) before typechecking, and they are only needed when estimating a function application's cost before the CPS conversion. For the cost model before code generation, C_{App} is simply defined as

$$\begin{aligned} C_{\mathsf{App}}^{\mathsf{PreCodeGen}} &= 2 \times C_{\mathsf{Var}} + C_{\mathsf{SetReg}} + C_{\textsc{Jump}} \\ \text{where } C_{\mathsf{Var}} &= C_{\mathsf{GetReg}} \\ C_{\mathsf{GetReg}} &= C_{\textsc{Push}} + C_{\textsc{MLoad}} \\ C_{\mathsf{SetReg}} &= C_{\textsc{Push}} + C_{\textsc{MStore}}. \end{aligned}$$

I use superscripts PreCodeGen, PreCC, and PreCPS to indicate the stage, and omit

arguments for cost functions (such as n_k and b_k for $C^{\mathsf{PreCodeGen}}_{\mathsf{App}}$) if they are not used. A cost parameter should actually be a pair for both time and memory. I stick to the convention from previous chapters that operations on numbers like + can be overloaded to mean operations on pairs. Curious readers can make sense of this cost by checking the code generation for function applications in Figure 4-11.

For an abstraction (or "closure creation"), the two parameters that determine its cost are $C_{\mathsf{AbsInner}}(e)$ and $C_{\mathsf{Abs}}(e)$, which are defined at this stage as

$$\begin{aligned} C^{\mathsf{PreCodeGen}}_{\mathsf{AbsInner}} &= C_{\textsc{JumpDest}} \\ C^{\mathsf{PreCodeGen}}_{\mathsf{Abs}} &= (0, 0). \end{aligned}$$

C_{AbsInner} is for the prelude and epilogue code added to the function body by the transformation. This cost will occur when the function is invoked, so it is added to the cost index in the arrow type. On the other hand, C_{Abs} is for the creation of the abstraction (closure), occurring at the function's defining site. An abstraction has zero cost at this post-closure-conversion stage because here every function is a top-level definition[54].

The cost models of index/type application and index/type polymorphism (or index/type abstraction) are similar to those for function application and lambda abstraction. I will focus on type application and abstraction here since the index version is almost the same. The first thing to notice is that the ∀ type is also indexed by a cost, akin to the arrow type, because a type abstraction also needs some computation to produce its result in the pre-CPS-conversion stage. But in the pre-code-generation stage, all cost parameters are zero:

$$\begin{aligned} C^{\mathsf{PreCodeGen}}_{\mathsf{AppT}} &= (0, 0) \\ C^{\mathsf{PreCodeGen}}_{\mathsf{AbsTInner}} &= (0, 0) \\ C^{\mathsf{PreCodeGen}}_{\mathsf{AbsT}} &= (0, 0). \end{aligned}$$

[54] And function names are represented as code labels, not stored to registers.

For case analysis, the single cost parameter C_{Case} is

$$\begin{array}{rcl} C^{\mathsf{PreCodeGen}}_{\mathsf{Case}}(i_1, i_2) & = & C_{\mathsf{CaseCommon}} + \max(i_1, i_2 + C_{\textsc{JumpDest}}) \\ \text{where } C_{\mathsf{CaseCommon}} & = & C_{\mathsf{Var}} + C_{\textsc{Push}} + C_{\textsc{BrSum}} + C_{\textsc{Push}} + C_{\textsc{Add}} + C_{\textsc{MLoad}} + C_{\mathsf{SetReg}}. \end{array}$$

Note that the second branch has an extra cost $C_{\textsc{JumpDest}}$ because of the JumpDest instruction at the beginning of each code block. The other branching expressions have similar cost definitions:

$$\begin{array}{rcl} C^{\mathsf{PreCodeGen}}_{\mathsf{If}}(i_1, i_2) & = & C_{\mathsf{IfCommon}} + \max(i_1, i_2 + C_{\textsc{JumpDest}}) \\ C^{\mathsf{PreCodeGen}}_{\mathsf{Ifi}}(i_1, i_2) & = & C_{\mathsf{IfiCommon}} + \max(i_1, i_2 + C_{\textsc{JumpDest}}) \\ \text{where } C_{\mathsf{IfCommon}} & = & C_{\mathsf{Var}} + C_{\textsc{IsZero}} + C_{\textsc{Push}} + C_{\textsc{JumpI}} \\ \text{where } C_{\mathsf{IfiCommon}} & = & C_{\mathsf{Var}} + C_{\textsc{IsZero}} + C_{\textsc{Push}} + C_{\textsc{Swap}} + C_{\textsc{Push}} + C_{\textsc{JumpI}} + C_{\mathsf{SetReg}}. \end{array}$$

4.5.3 TiML cost model before closure conversion

The cost of a function application before closure conversion, as shown below, is only a bit larger than that before code generation. The extra costs correspond to the few operations closure conversion inserts for a function application in Figure 4-5.

$$\begin{array}{rcl} C^{\mathsf{PreCC}}_{\mathsf{App}} & = & C^{\mathsf{PreCodeGen}}_{\mathsf{App}} + C_{\mathsf{Unpack}} + 2 \times C_{\mathsf{Proj}} + C_{\mathsf{Pair}} \\ \text{where } C_{\mathsf{Pair}} & = & C_{\mathsf{Tuple}}(2) \end{array}$$

The costs of an abstraction at this stage start to depend on the function body. More precisely, they depend on the number of free term variables in the function body, which is understandable since the main job of closure conversion is to collect these free variables into a tuple. The costs are defined below:

$$\begin{array}{rcl} C^{\mathsf{PreCC}}_{\mathsf{AbsInner}}(e) & = & C^{\mathsf{PreCodeGen}}_{\mathsf{AbsInner}} + 2 \times C_{\mathsf{Proj}} + C_{\mathsf{Pair}} + C_{\mathsf{Pack}} + |FV(e)| \times (C_{\mathsf{Let}} + C_{\mathsf{Proj}} + C_{\mathsf{Var}}) \\ C^{\mathsf{PreCC}}_{\mathsf{Abs}}(e) & = & C_{\mathsf{Tuple}}(|FV(e)|) + C_{\mathsf{Pair}} + C_{\mathsf{Pack}}. \\ \text{where } C_{\mathsf{Let}} & = & C_{\mathsf{SetReg}} \end{array}$$

As in Figure 4-5, $FV(e)$ stands for the set of free term variables in e. The cost of creating an abstraction, represented by C_{Abs}, comes from the creation of a tuple of

length $|FV(e)|$ and some other objects (a pair and a existential package), while the cost of the prelude code added to the function body by closure conversion involves the projection of each field of the environment tuple and other administrative operations.

The costs of type application and type abstraction are shown below. Remember that the CPS conversion converts each type abstraction into a lambda abstraction (surrounded by some index/type abstractions), so at this post-CPS-conversion stage, type applications are only for typechecking purposes and do not have runtime effects, which is why C_{AppT} is zero. Closure conversion skips over type abstractions (only modifies lambda abstractions), so $C_{\mathsf{AbsTInner}}$ is also zero. The cost of creating a type abstraction is i, the cost of the body, because after CPS conversion the body is always a lambda abstraction (surrounded by some index/type abstractions), and this $C_{\mathsf{AbsT}}(i) = i$ is just for relaying the cost of creating the lambda abstraction to the outside world.

$$\begin{aligned} C^{\mathsf{PreCC}}_{\mathsf{AppT}} &= (0,0) \\ C^{\mathsf{PreCC}}_{\mathsf{AbsTInner}} &= (0,0) \\ C^{\mathsf{PreCC}}_{\mathsf{AbsT}}(i) &= i \end{aligned}$$

The costs for branching are exactly the same as in the pre-code-generation stage, since closure conversion does not perform any special transformations on branching.

4.5.4 TiML cost model before CPS conversion

The effects of CPS conversion are best illustrated by the cost of function applications, shown below.

$$\begin{aligned} C^{\mathsf{PreCPS}}_{\mathsf{App}}(n_k, b_k) &= C^{\mathsf{PreCC}}_{\mathsf{App}} + C_{\mathsf{Pair}} + C_{\mathsf{CreateK}}(n_k, b_k) \\ \text{where } C_{\mathsf{CreateK}}(n_k, b_k) &= b_k \ ?\ C^{\mathsf{PreCC}}_{\mathsf{Abs}}(n_k) + C^{\mathsf{PreCC}}_{\mathsf{AbsInner}}(n_k) : (0,0) \end{aligned}$$

Compared with $C^{\mathsf{PreCC}}_{\mathsf{App}}$, $C^{\mathsf{PreCPS}}_{\mathsf{App}}$ contains the extra cost of creating a pair (to contain the argument and the continuation) and potentially creating a function as the continuation. Looking at the CPS conversion for function applications in Figure 4-4, readers may wonder why a function needs to be created when the continuation k

should already be a function[55]. The reason is that when k is used in an application $k\ e$ in the translated code and k is known to be $\lambda x.\ e'$, $k\ e$ is replaced by $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ (i.e. creating and applying a function is replaced by a let-binding) to avoid the cost of function creation and application. The only case when k appears in the translated code as an explicit abstraction is when k is assigned to a variable[56], in which case the cost of creating an abstraction will occur.

Whether a continuation function will be created depends on whether there is a continuation: if there is no continuation, which in a CPS-ed world means the continuation is just the continuation variable passed in as an argument, no new function will be created since the variable will be used as the continuation; otherwise a function will be created. This function will be invoked exactly once, so the cost of creating this function is $C^{\mathsf{PreCC}}_{\mathsf{Abs}}(n_k) + C^{\mathsf{PreCC}}_{\mathsf{AbsInner}}(n_k)$. I use n_k here as the argument because $C^{\mathsf{PreCC}}_{\mathsf{Abs}}(e)$ and $C^{\mathsf{PreCC}}_{\mathsf{AbsInner}}(e)$ really only depend on $|FV(e)|$, and n_k is equal to $|FV(k)|$, the number of free variables in the continuation after the call. In other words, the cost of calling a function at the pre-CPS-conversion stage depends on the number of live variables after the call. This cost is avoided when the call is a tail call, which reflects the fact that CPS conversion automatically performs tail-call optimization[57].

For the costs of an abstraction (shown below), a subtlety is that there could be a function call added by CPS conversion at the end of the function body. This is the call to the continuation at the end: $k\ (\cdots)$. This call is avoided if the function body ends with a function/index/type application or a branch. $\mathsf{hasTailCall(e)}$ denotes this test.

$$
\begin{aligned}
C^{\mathsf{PreCPS}}_{\mathsf{AbsInner}}(e) &= C^{\mathsf{PreCC}}_{\mathsf{AbsInner}}(e) + 2 \times C_{\mathsf{Proj}} + C_{\mathsf{CallK}}(e) \\
C^{\mathsf{PreCPS}}_{\mathsf{Abs}}(e) &= C^{\mathsf{PreCC}}_{\mathsf{Abs}}(e) \\
\text{where}\ \ C_{\mathsf{CallK}}(e) &= \mathsf{hasTailCall(e)}\ ?\ C^{\mathsf{PreCC}}_{\mathsf{App}} : (0,0)
\end{aligned}
$$

[55]The CPS conversion maintains an invariant that k is either a lambda abstraction or a variable.

[56]In the case here (of translating a function application), k is used as a function argument in the translated code, which is the same as being assigned to a variable, because the translated code is let-normalized after CPS conversion.

[57]More generally, CPS conversion automatically performs a "frame compression" at each function call, in the sense that instead of backing up the whole call-frame for restoring local variables in the future (which is what C does), a function call implemented by CPS conversion only selectively backs up those local variables that will actually be used after the call.

The costs of type application and abstraction (shown below) are very similar to those for functions, with the only difference being the absence of C_{Pair} and $2 \times C_{\mathsf{Proj}}$. The similarity is because the overheads introduced by CPS conversion, such as creating the continuation function and the extra $k\ (\cdots)$ at the end of the function body, are the same in both cases.

$$
\begin{aligned}
C^{\mathsf{PreCPS}}_{\mathsf{AppT}}(n_k, b_k) &= C^{\mathsf{PreCC}}_{\mathsf{App}} + C_{\mathsf{CreateK}}(n_k, b_k) \\
C^{\mathsf{PreCPS}}_{\mathsf{AbsTInner}}(e) &= C^{\mathsf{PreCC}}_{\mathsf{AbsInner}}(e) + C_{\mathsf{CallK}}(e) \\
C^{\mathsf{PreCPS}}_{\mathsf{AbsT}}(e) &= C^{\mathsf{PreCC}}_{\mathsf{Abs}}(e)
\end{aligned}
$$

The costs of branches before CPS conversion are more involved than the versions at other stages because CPS conversion (as in Figure 4-4) assigns the continuation to a variable before branching and uses the variable as continuation when transforming the branches (to avoid code duplication). Assigning the continuation to a variable could potentially incur a function creation as discussed above. These extra costs are easy to understand at this point since they are C_{CreateK} and C_{CallK}, exactly the same overheads as in applications and abstractions. The reason for adding them is the same as before.

$$
\begin{aligned}
C^{\mathsf{PreCPS}}_{\mathsf{Case}}(i_1, i_2, e_1, e_2, n_k, b_k) &= C_{\mathsf{CaseCommon}} + C_{\mathsf{CreateK}}(n_k, b_k) \\
&\quad + \max(i_1 + C_{\mathsf{CallK}}(e_1),\ i_2 + C_{\textsc{JumpDest}} + C_{\mathsf{CallK}}(e_2))
\end{aligned}
$$

The other branching expressions have similar cost models:

$$
\begin{aligned}
C^{\mathsf{PreCPS}}_{\mathsf{If}}(i_1, i_2, e_1, e_2, n_k, b_k) &= C_{\mathsf{IfCommon}} + C_{\mathsf{CreateK}}(n_k, b_k) \\
&\quad + \max(i_1 + C_{\mathsf{CallK}}(e_1),\ i_2 + C_{\textsc{JumpDest}} + C_{\mathsf{CallK}}(e_2)) \\
C^{\mathsf{PreCPS}}_{\mathsf{Ifi}}(i_1, i_2, e_1, e_2, n_k, b_k) &= C_{\mathsf{IfiCommon}} + C_{\mathsf{CreateK}}(n_k, b_k) \\
&\quad + \max(i_1 + C_{\mathsf{CallK}}(e_1),\ i_2 + C_{\textsc{JumpDest}} + C_{\mathsf{CallK}}(e_2)).
\end{aligned}
$$

4.5.5 Surface-TiML cost model

The gap between Surface-TiML and μTiML is mainly about datatypes, so the only cost parameters that are Surface-TiML-specific are those for constructor applica-

tions and pattern matchings. The cost of a constructor application is determined by $C_{\mathsf{Constr}}(k, n, m)$ (see Figure 4-2), defined below, where k is the number of index arguments, n is the number of constructors the datatype has, and m is the position of c among its siblings (starting from 0).

$$C_{\mathsf{Constr}}(k, n, m) = (k + 1) \times C_{\mathsf{Pack}} + \mathsf{numInj}(m, n) \times C_{\mathsf{Inj}} + C_{\mathsf{Fold}}$$

$\mathsf{numInj}(m, n)$ calculates the number of left/right injections it needs to implement a n-way injection $\mathsf{inj}^{m,n}$ (see Equation 4.1.2.3), which is defined as

$$\begin{array}{rcl} \mathsf{numInj}(0, 1) & = & 0 \\ \mathsf{numInj}(0, n) & = & 1 \\ \mathsf{numInj}(m, n) & = & 1 + \mathsf{numInj}(m - 1, n - 1). \end{array}$$

It is easy to see that this definition is derived from Equation 4.1.2.3.

The cost of compound pattern matching is hard to express as simple mathematical formulas, since the transformation (described in Section 4.1.2) is somewhat involved. The Surface-TiML typechecker employs a "simulation" strategy: when typechecking a pattern matching, it does a dry-run of the translation described in Section 4.1.2 to find out how many projections/unfolds/unpackings/case-analyses will be added to each branch[58], and then uses this information to calculate the cost overhead of each branch. Note that this strategy is essentially "compile-then-typecheck," which is against this thesis's general principle that typechecking (including cost estimation) should happen before compilation. But this violation is localized (only for estimating the costs of compound pattern matchings), and the version of the transformation used as a subroutine of the typechecking (the dry-run) is simpler and faster than the real transformation, so I think it is acceptable.

[58] The dry-run is faster than the real transformation because it uses "fake expressions" as the result which only record the projections/unfolds/unpackings/case-analyses introduced by the transformation (i.e. they are just the skeletons of the real results).

Chapter 5

Evaluation

The evaluation of this thesis consists of two parts. In the first part, I use TiML to prove the time complexities of 17 classic algorithms in order to gauge the expressivity of the language and the user experience (e.g. typechecking speed, annotation burden, etc.) of the system. In the second part, I use TiML to typecheck and compile 8 real-world smart contracts, demonstrating that TiML can give static estimations of their actual gas costs (using the official EVM cost model) that are both sound (estimated costs are no less than the real costs) and accurate (estimated costs are close to the real costs), and that gas costs of TiML-generated code are comparable to those of Solidity [8], today's mainstream smart-contract language/compiler.

5.1 Typechecking classic algorithms

I have tested the TiML typechecker on 17 benchmarks, incorporating classic data structures and algorithms including trees, doubly linked lists, insertion sort, array-based merge sort (copying and in-place), quicksort, binary search, array-based binary heap, k-median search, red-black trees, Braun trees, Dijkstra's algorithm for graph shortest paths, functional queues (amortized analysis), and dynamic tables (amortized analysis). The cost model used in this section is the simple cost model described in Section 2.1 where only function application costs one unit of time, all other operations and all memory usage being free.

The test is run on a 2.5-GHz quad-core Intel Core i7 CPU with 16GB RAM (actual memory usage is within 256MB). The SMT solver I use is Z3 [33] 4.4.1. Table 5.1 lists each benchmark's filename, description, total time of typechecking (including time for parsing, typechecking, inference, and VC solving), lines of code, lines of code that contain time annotations, and asymptotic complexities of the most representative top-level functions. The numbers are also illustrated in Figure 5-1 and 5-2. Every benchmark finishes within 0.5 second, most of them within 0.3 seconds.

Name	Description	Time (s)	LoC	LoC with Anno	A. Comp.
list	List operations	0.155	48	9	n, mn
ragged-matrix	Ragged matrices	0.113	16	1	m^2n
tree	Trees	0.18	86	10	mn
msort	Merge sort	0.221	49	8	$mn\log n$
insertion-sort	Insertion sort	0.142	25	4	mn^2
braun-tree	Braun trees	0.199	98	11	$\log n$, $\log^2 n$
rbt	Red-black trees	0.422	316	19	$\log n$
dynamic-table	Dynamic tables	0.153	126	10	(amortized) 1
functional-queue	Functional queues	0.137	95	6	(amortized) 1
array-bsearch	Binary search	0.149	44	2	$m\log n$
array-heap	Binary heap	0.221	139	6	$m\log n$
array-msort	Merge sort on arrays	0.228	112	7	$mn\log n$
array-msort-inplace	In-place merge sort on arrays	0.255	133	9	mn^2
array-kmed	k-median search	0.16	70	8	mn^2
dlist	Doubly linked lists	0.26	112	10	mn
qsort	Quicksort	0.128	43	7	mn^2
dijkstra	Dijkstra's alg. (shortest paths)	0.12	75	0	$(m_+ + m_\leq)n^2$

Table 5.1: Benchmarks. Columns show total time of typechecking (including parsing, typechecking, inference, and VC solving), lines of code, lines of code containing time annotations, and asymptotic complexities of the most representative top-level functions.

As an empirical study of the usability of TiML, I explain each benchmark and analyze the annotations in it. Most of the annotations are at two places: the types of recursive functions and pattern matches. The former is akin to pre/post-conditions and loop invariants in program logics. The latter sometimes require annotations because of the "forgetting problem" of existential-type eliminations: the running

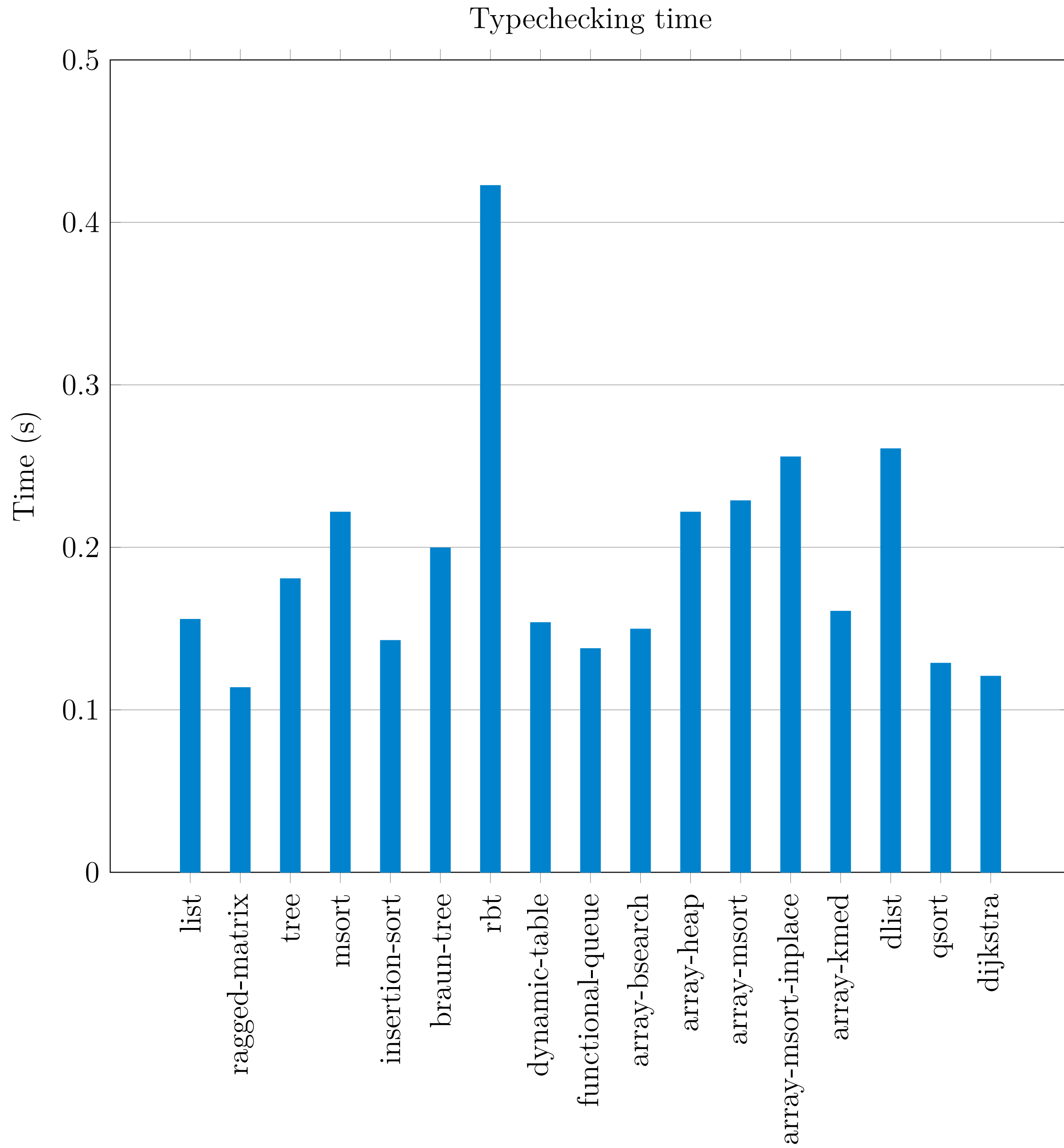

Figure 5-1: Typechecking time. This graph shows the total time of typechecking (including parsing, typechecking, inference, and VC solving) for each benchmark program. Every benchmark finishes within 0.5 second, most of them within 0.3 seconds.

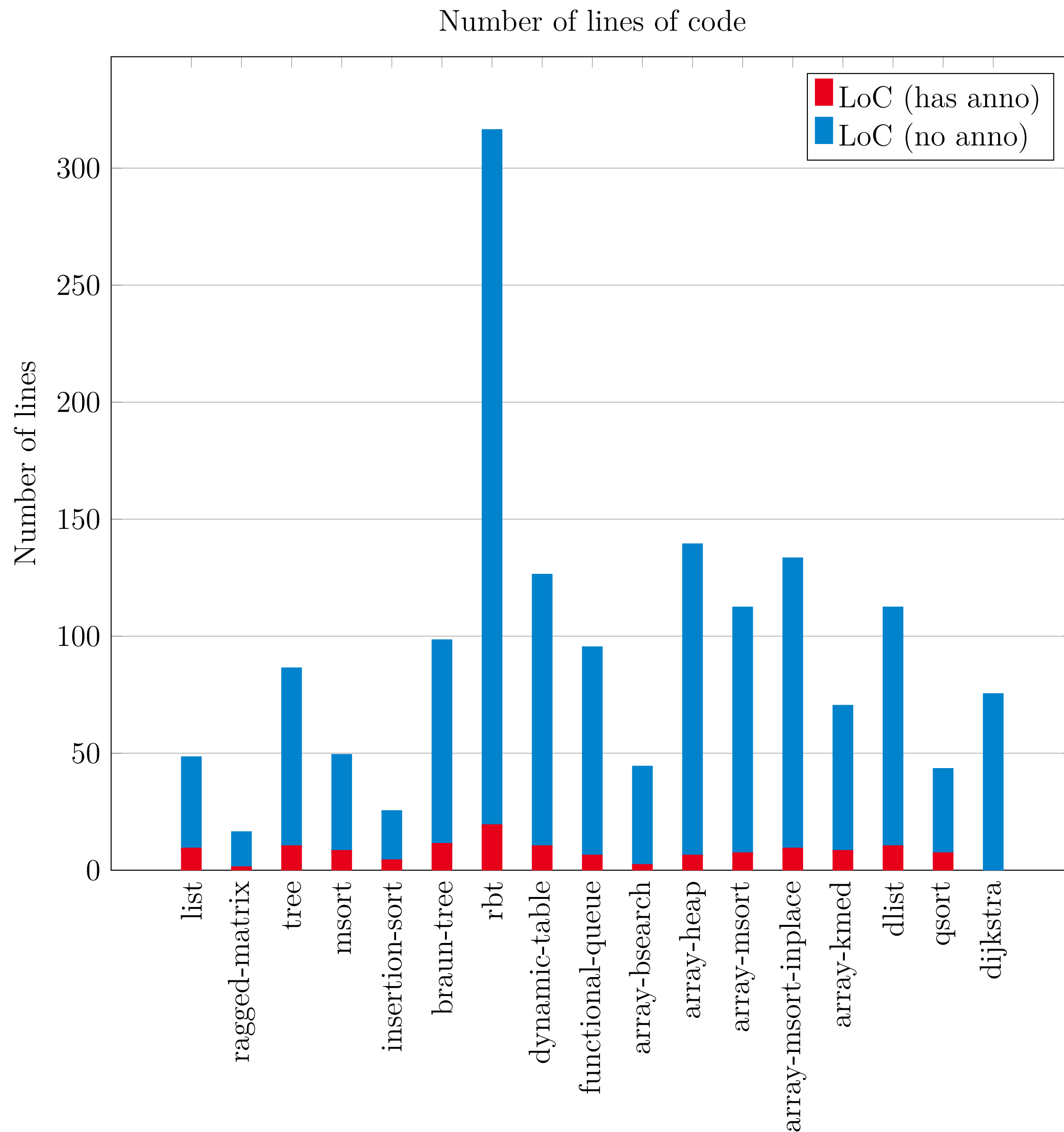

Figure 5-2: Number of lines of code. The upper part of each bar represents the number of code lines without annotations, while the lower part represents those that contain annotations. The ratio of the lower part to the whole bar reflects the annotation burden.

time and result type of `unpack` should not contain the locally introduced type/index variables, but it can be hard for the typechecker to figure out how to forget them. I allow `using` and `return` clauses on `case` to specify the common running time and result type of all branches, which cannot reference branch-local variables. It is similar to the `return` clauses of dependent pattern-matching in Coq. I use some syntactical tricks to guess these clauses. For example, if a case analysis is directly under a recursive function, I copy the `using` and `return` annotations in the recursive-function signature.

All annotations in benchmark `list` (list operations) are types of recursive functions similar to `foldl`. Benchmark `ragged-matrix` contains lists of lists with one index being the length of the outer list and another index being the maximal length of the inner lists. Benchmark `tree` contains binary trees and operations on them such as map, fold, and flatten. The annotations in these two files are similar to those in `list`. Benchmark `msort` has been discussed in Section 2.1. Benchmark `insertion-sort` is specified using big-O similarly to `msort`.

Benchmark `braun-tree` contains Braun trees [65], a kind of balanced binary trees for functional implementation of priority queues. In a Braun tree, each node stores a value that is smaller than all values in the children, and the size of the left child is equal to or larger by one than that of the right child. It supports enqueue and dequeue in $O(\log n)$ and $O(\log^2 n)$ time respectively. I define Braun trees as binary trees indexed by size (i.e. number of nodes), and for a Braun tree of size $n+1$, I require the sizes of its left and right child to be $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ respectively. All functions in `braun-tree` are specified with big-O complexities, so the time-annotation burden is on par with `msort`'s. In this benchmark, some implicit-index-argument inference failed, so I have to supply index arguments explicitly using the **`@fun_name`** syntax (similar to Coq). In benchmark `rbt` for red-black trees, I have to put two extra invariants in the definition of `rbt` other than those shown in Section 2.1. These two invariants can be derived from the other invariants, but because in TiML lemmas, lemma invocations, and inductions must be written as ordinary functions (like in Dafny) which increase a program's running time, deriving these invariants on-the-fly

will increase asymptotic complexity. Annotation burden for time is again on par with msort's since big-O complexities are used. In the jump from black-height to logarithm of tree size, an assumed lemma is used relating logarithms and exponentials.

Benchmarks functional-queue and dynamic-table are two examples showing how to use TiML to conduct amortized analysis. In a standard cost analysis, a function's time is specified in terms of only the input size. Let us write such a type as $\forall n.\ \tau_n \xrightarrow{g(n)} \tau'$. In the "potential method" for doing amortized complexity analysis [27], the running time c is specified by an inequality $c + \Phi_1 \leq c_a + \Phi_0$. Here Φ is a (nonnegative) *potential function* defined on configurations (i.e. states), and Φ_0 and Φ_1 are the potentials before and after the function. c_a is called "amortized cost." We can write the type of such a function as $\forall n.\ \exists c\ n'.\ \tau_n \xrightarrow{c} \tau_{n'} \wedge P(n, c, n')$, where $P(n, c, n') \stackrel{\text{def}}{=} c + \Phi(n') \leq c_a + \Phi(n)$ (c_a is a parameter). Because sometimes c and n' need to depend on the input value (not just its type), the existential quantifiers need to be pushed later: $\forall n.\ \tau_n \xrightarrow{k} \exists c.\ \mathsf{unit} \xrightarrow{c} \exists n'.\ \tau_{n'} \wedge P(n, c, n')$, where k is a constant. I call type $\exists n'.\ \tau_{n'} \wedge P(n, c, n')$ some_output_and_cost_constraint and type $\exists c.\ \mathsf{unit} \xrightarrow{c} \exists n'.\ \tau_{n'} \wedge P(n, c, n')$ amortized_comp ("amortized computation") in the TiML code.

A functional queue [66] is a queue implemented by two stacks, one for receiving input, the other for supplying output. When the output stack is empty, the content in the input stack is dumped to the output stack, in reverse. A dynamic table [27] (like the "vector" container in C++'s STL) is a dynamically allocated buffer that enlarges itself when the load factor becomes too high after an insertion, and shrinks itself when the load factor becomes too low after a deletion. Both of these two data structures enjoy amortized constant-time insertion and deletion. Note that TiML does not have any built-in support for amortized analysis, so being able to do it somewhat surprised me. It is in line with many language designers' experience that when one starts with primitives to encode the most basic concepts, many computational phenomena will arise naturally.

Benchmarks array-bsearch, array-heap, array-msort, and array-msort-inplace are array-based implementations for binary search, binary heaps, and merge sort

(copying and in-place). Their time-annotation burdens are on par with `foldl` and `msort`'s. Benchmark `array-kmed` does k-median search on an array. Big-O inference fails in this benchmark, so precise bounds are given. The VC that fails the Big-O inference is $15+m+\max(T(m,n-1),T(m,n-1)+4) \leq T(m,n)-8$. The culprit is the "$-8$" on the right-hand side. The Big-O inferrer only recognizes a recurrence whose right-hand side is $T(\dots)$. I can add annotations to massage the VC by moving "-8" to the left-hand side, which may or may not be better than just spelling out $T()$ in this case. Function `array_kth_median_on_range` uses local time annotation to forget a local index variable. Benchmark `dlist` implements doubly linked lists using references (i.e. arrays). Each function just needs one big-O annotation. Benchmark `qsort` for quicksort requires a rather detailed time annotation for function `list_qsort` to forget the two local index variables that are the lengths of the two partitions. The running time of `list_qsort` is first calculated in terms of these two lengths, and it is very hard for the typechecker to figure out how to replace these two lengths with the total length of the input list, hence the annotation. Benchmark `dijkstra` implements Dijkstra's algorithm for calculating shortest paths, which surprisingly does not require any time annotation. The reason is that the algorithm is implemented by mainly using standard iterators such as `app`, `appi`, and `foldli` provided by the `Array` module in the standard library, demonstrating the power of modularity preserved by TiML from SML.

Among the benchmarks, `braun-tree`, `rbt`, `dynamic-table`, `functional-queue`, and all the array-based algorithms crucially rely on the refinement mechanism to encode data-structure invariants and algorithm preconditions.

As an empirical data point, an undergraduate student with background in SML took just one day to become fluent in writing and annotating TiML programs.

5.2 Compiling smart contracts

I collected 8 representative smart contracts as a benchmark set, which cover most of the current and hypothetical use cases of smart contracts. The contract names are

listed in Table 5.2, each with a short description. These contracts are collected from Solidity's tutorial and the Ethereum blockchain. For the latter, I chose the contracts that had the highest numbers of transactions (i.e. most used) or the largest balances (i.e. managing the most wealth) at the time of collection.

Contract	Description	Source
Token	A fixed-supply asset that can be transferred between accounts.	Solidity Tutorial
CrowdSale	A contract the allows any person to buy any amount of a pre-specified token with ethers at a pre-specified price. It is designed for Initial Coin Offerings (ICO).	Solidity Tutorial
EtherDelta	A token exchange that allows people to buy and sell different kinds of tokens (prices are determined by bidding, as in stock exchanges).	Ethereum blockchain
Congress	A voting congress that decides on proposals by voting.	Solidity Tutorial
MultiSig	A multi-sig wallet that is owned by multiple people and requires a specified number to owners to approve expenses that exceed a daily limit.	Ethereum blockchain
CryptoKitties	A collectible non-fungible token ("kitties") where each kitty has a unique "gene" and two kitties can mate to produce a new kitty. Kitties can be auctioned for their ownership or right to sire.	Ethereum blockchain
BlindAuction	An auction where each bid is concealed (by allowing the bid to be fake and hiding the fakeness behind a hash) and only after the deadline will all real bids be revealed. It is useful for revealing a bidder's true preference since she is shielded from the interference of other bidders.	Solidity Tutorial
SafeRemotePurchase	A contract for online purchase where both the seller and the buyer deposit ethers twice the value of the goods into the contract and can only get their deposits back when the buyer confirms receipt of the goods. The seller is disincentivized from withholding the goods (not shipping them), and the buyer is disincentivized from not confirming receipt, because in doing so they will lose more than the value of the goods[1].	Solidity tutorial

Table 5.2: Benchmark contracts and their descriptions. The last column shows where each contract's source code comes from.

[1] If the buyer refuses to confirm after he has received the goods, he will lose v while the the seller will lose $3v$ (v is the value of the goods). So a malicious buyer can impose larger damage on the seller than on himself.

All 8 contracts were written in Solidity. I rewrote them in ETiML and took care to make sure that they closely mimic the original Solidity ones[2]. After typechecking and compiling them, I ran the generated bytecode programs (in binary format) on geth [6], the official Ethereum implementation in the Go language [7], which can report the actual gas usage of the bytecode.

I use two measurements as the main results of evaluation in this section: (1) the ratio of the actual gas cost to the costs estimated by the type system, to show that the estimation is both sound (ratio not larger than 1) and accurate (ratio close to 1); (2) the ratio of the gas cost of a contract compiled by the TiML compiler to the gas cost of the one compiled by the Solidity compiler (the "slow-down" factor), to show that the code generated by my compiler is reasonably efficient despite the fact that it does not have any nontrivial optimizations and was designed mainly for cost-estimation soundness and accuracy. I treat a slow-down between 1x and 10x against Solidity-generated code as a sign that the system developed in this thesis is a feasible approach and could become practical when serious engineering effort on optimization is put into it.

Because each (public) function of a smart contract can be invoked individually, the unit of evaluation in this section is a function. Among the 8 benchmark contracts I collected 36 functions that are nontrivial enough and worth measuring. Some of them are different runs of the same function where the control flow takes different paths. Table 5.3 lists the scenarios in which the functions are invoked.

In measuring the first metric (estimation accuracy), I turned off EVM's special cost policy regarding zeroness in storage writes. As discussed in Chapter 3, writing a nonzero value to a storage address whose current value is zero costs significantly more than storage writes in other cases. Because TiML currently does not track the zeroness of each storage address (except for icell)[3] and can only assume the worst case, its static-estimation accuracy will be greatly affected by the zeroness overestimation.

[2] ETiML was designed with Solidity as the reference, so this rewriting is relatively straightforward.

[3] TiML's index language is actually almost expressive enough to represent the zeroness of each element of a map or a vector, since it has the "map index" $\{\overrightarrow{u : i}\}$. In the future I will extend the state specification to allow using a map index to specify the zeroness of each element of a vector or map (instead of just using a number to specify a vector's size).

Contract	Function	Scenario
Token	transfer	
	transferFrom	
	burnFrom	
	approve	
	burn	
	approveAndCall	
CrowdSale	default	
	checkGoalReached	
	safeWithdrawal-1	safeWithdrawal, goal not reached, amount>0
	safeWithdrawal-2	safeWithdrawal, goal reached
EtherDelta	trade	
	availableVolume	
Congress	addMember	
	removeMember	8 members (not including 0 and owner) added, remove the first
	newProposal	jobDescription="job", transaction-Bytecode="code"
	vote	justificationText="LGTM"
	executeProposal-1	executeProposal, passed
	executeProposal-2	executeProposal, not passed
MultiSig	confirmAndCheck-1	confirmAndCheck, 5 owners, first confirm, not enough
	confirmAndCheck-2	confirmAndCheck, 5 owners, second confirm, not enough
	confirmAndCheck-3	confirmAndCheck, 5 owners, third confirm, enough and passed
	revoke	
	clearPending	5 pending
	reorganizeOwners	5 original owners with the 2nd removed, reorganize
CryptoKitties	_createKitty	
	breedWithAuto	breeding two gen-1 kitties
	givenBirth	
	ClockAuction.createAuction	
	ClockAuction.bid	
	SaleClockAuction.bid	
	averageGen0SalePrice	
	tokensOfOwner	7 tokens in total, all belonging to the caller except the 3rd and 6th
BlindAuction	bid	
	reveal	5 bids, each higher than the previous one
SafeRemotePurchase	confirmPurchase	
	confirmReceived	

Table 5.3: The functions that have been measured and scenarios in which the functions are invoked.

To leave out this factor and measure the accuracy for all the other factors, I modified both my typechecker and geth to set the cost of zero-to-nonzero flip to be the same as for other storage writes. The modified geth is only used in this measurement.

For the comparison between TiML and Solidity, I commented out function calls to external contracts to focus on the cost of each function itself. A program in both TiML or Solidity gets a prelude for dispatching to the desired function based on a signature within the input data and decoding input data into internal representations. They also use the same ABI for encoding function signatures and input data [9], so the comparison between them is fair.

The results are shown in Table 5.4. The first two columns give the name of the contract and the function. The third column is the ratio of actual gas costs to estimated ones. The fourth column is the ratio of gas costs of TiML-compiled code to those of Solidity. The accuracy and TiML-vs-Solidity results are also visualized in Figures 5-3 and 5-4.

As can be seen in Table 5.4, all cost estimations are sound in the sense that the accuracy is never above 1. In most cases, the accuracy is above 0.9. In each case where accuracy is below 0.9, the execution does not take the longest path, so the inaccuracy is mostly from overestimation for branching. Other minor sources of inaccuracy include overestimating the cost of reading a variable or a state name (a variable implemented by a code label is cheaper to read than one implemented by a register) and overestimation caused by a simple optimization that inlines some let-bindings (when the bound expression is e.g. type application[4], packing, folding, or unfolding). Note that there is a trade-off between estimation accuracy and code efficiency. More aggressive and complicated optimizations make the code more efficient but harder to estimate statically.

When compared against Solidity, most functions compiled by TiML see a slow-down of around 2x. There are two exceptions, function `averageGen0SalePrice` and `tokensOfOwner` in contract `CrypoKitties`, which suffer slow-downs of 17.795x and 11.89x. The main source of these large slow-downs is the implementation of for-loops

[4]Only after CPS conversion.

Contract	Function	Actual/Estimate	ETiML/Solidity
Token	transfer	0.976	1.452
	transferFrom	0.978	1.498
	burnFrom	0.981	1.616
	approve	0.993	1.093
	burn	0.981	1.629
	approveAndCall	0.981	1.121
CrowdSale	default	0.981	1.674
	checkGoalReached	0.988	1.629
	safeWithdrawal-1	0.728	1.084
	safeWithdrawal-2	0.437	1.258
EtherDelta	trade	0.974	1.540
	availableVolume	0.965	2.325
Congress	addMember	0.986	1.073
	removeMember	0.850	1.436
	newProposal	0.994	0.745
	vote	0.981	1.149
	executeProposal-1	0.988	1.596
	executeProposal-2	0.667	1.247
MultiSig	confirmAndCheck-1	0.846	1.073
	confirmAndCheck-2	0.382	1.651
	confirmAndCheck-3	0.519	1.229
	revoke	0.964	1.625
	clearPending	0.970	1.420
	reorganizeOwners	0.474	2.074
CryptoKitties	_createKitty	0.817	1.468
	breedWithAuto	0.983	2.342
	givenBirth	0.851	1.879
	ClockAuction.createAuction	0.984	1.659
	ClockAuction.bid	0.979	1.497
	SaleClockAuction.bid	0.982	1.207
	averageGen0SalePrice	0.911	**17.795**
	tokensOfOwner	0.919	**11.890**
BlindAuction	bid	0.997	1.013
	reveal	0.927	1.360
SafeRemotePurchase	confirmPurchase	0.985	1.652
	confirmReceived	0.988	1.068

Table 5.4: Evaluation results on the 8 benchmark smart contracts. The first two columns give the name of the contract and the function. The third column is the ratio of actual gas costs to estimated ones. The fourth column is the ratio of gas costs of TiML-compiled code to those of Solidity-compiled code (the "slow-down" factor). The largest two slow-downs are mainly caused by overhead of the `for` combinator from the TiML standard library, subtracting which will reduce the slow-downs to 1.972 and 3.67.

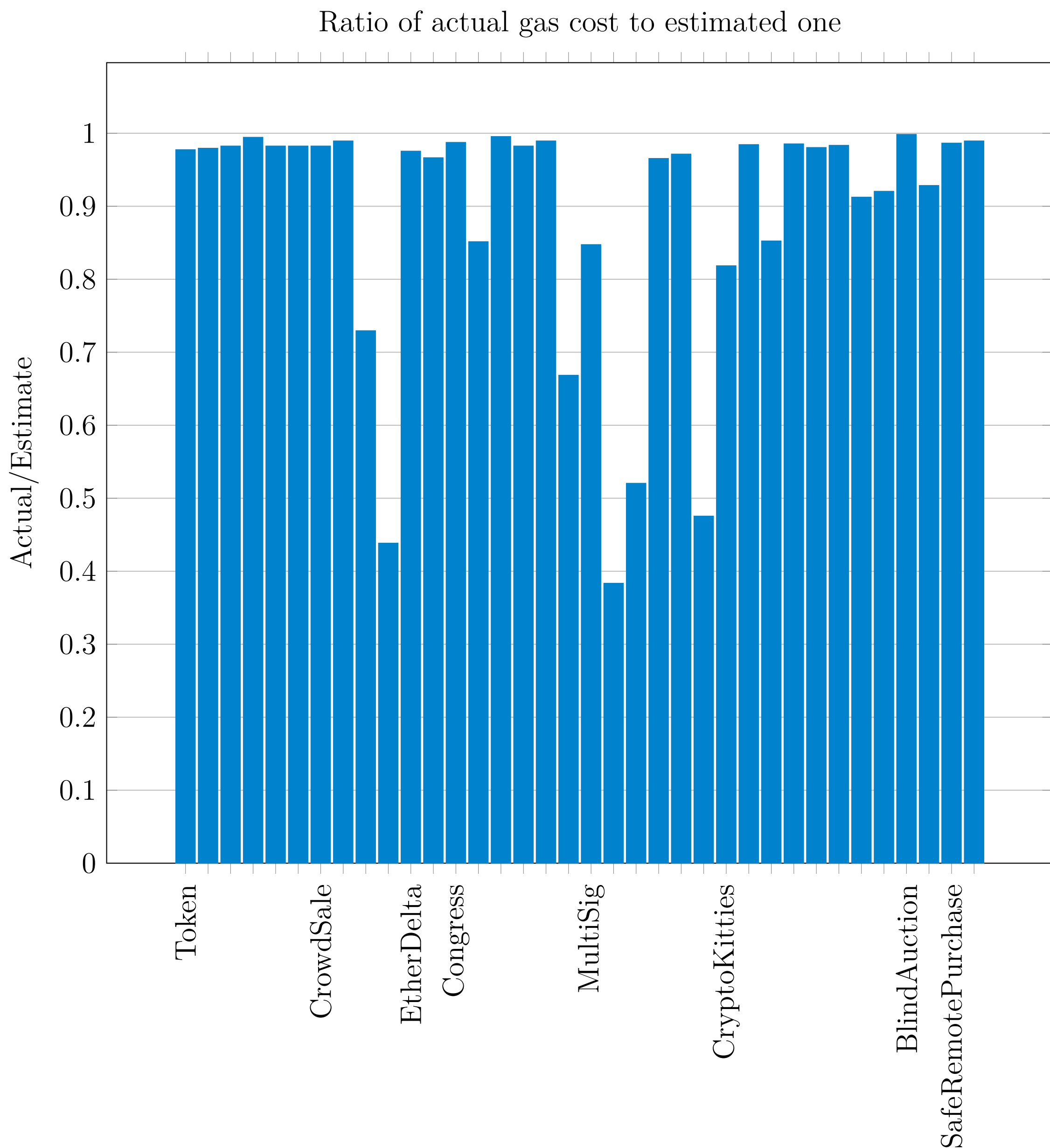

Figure 5-3: Gas-estimation accuracy. This graph shows the ratio of the actual gas cost to that estimated by the type system, for each function. The estimation is for upperbounds, whose soundness is witnessed by the fact that the ratios are never above 1. Each ratio below 0.9 is caused by the execution not taking the longest path. The cost of a zero-to-nonzero storage write is set to be the same as other storage writes in this measurement.

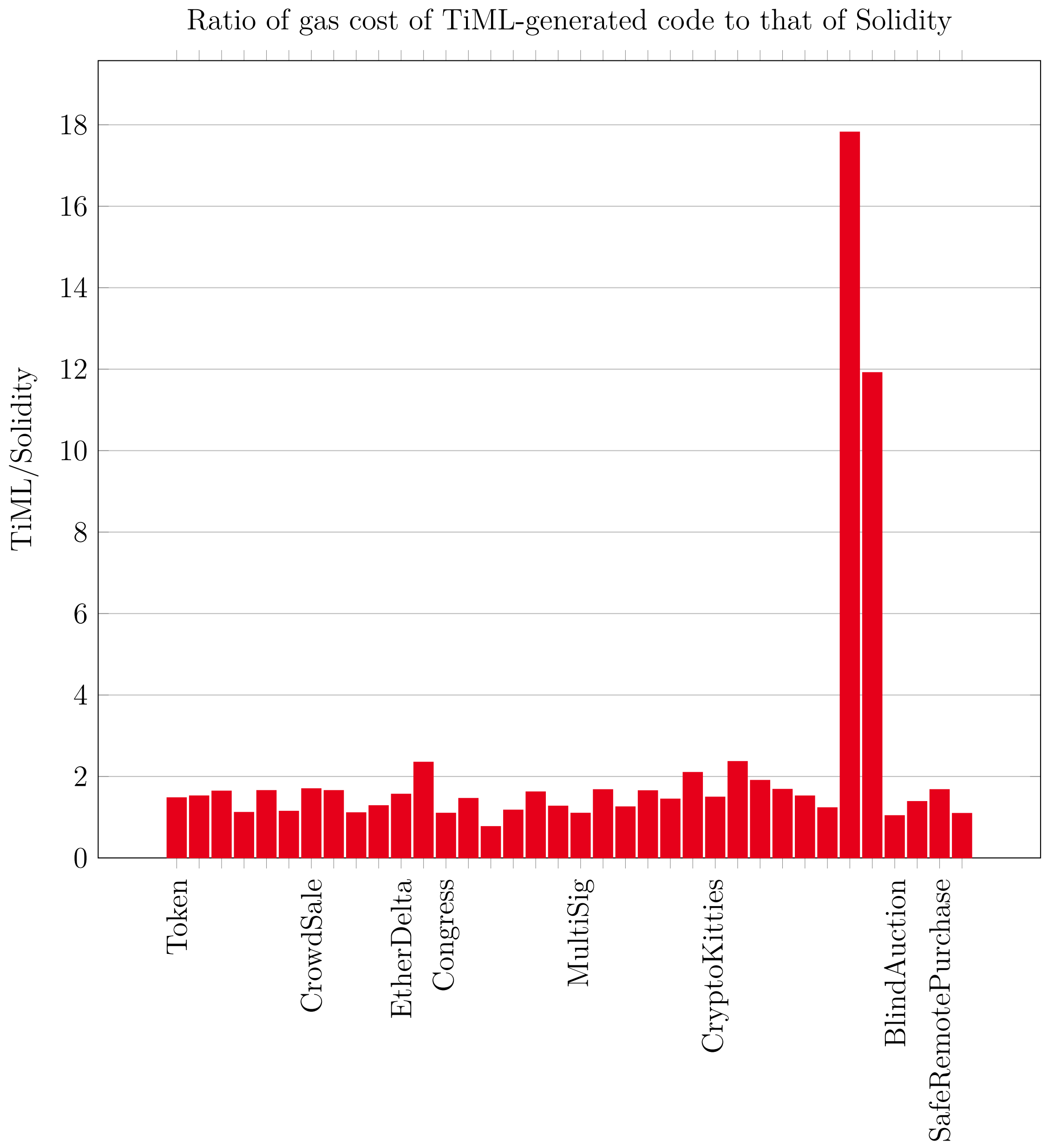

Figure 5-4: TiML vs. Solidity. This graph shows the ratio of the gas cost of TiML-compiled code to that of Solidity-compiled code (the "slow-down" factor) measured for each function. The largest two slow-downs are mainly caused by overhead of the `for` combinator from the TiML standard library, subtracting which will reduce the slow-downs to 1.972 and 3.67.

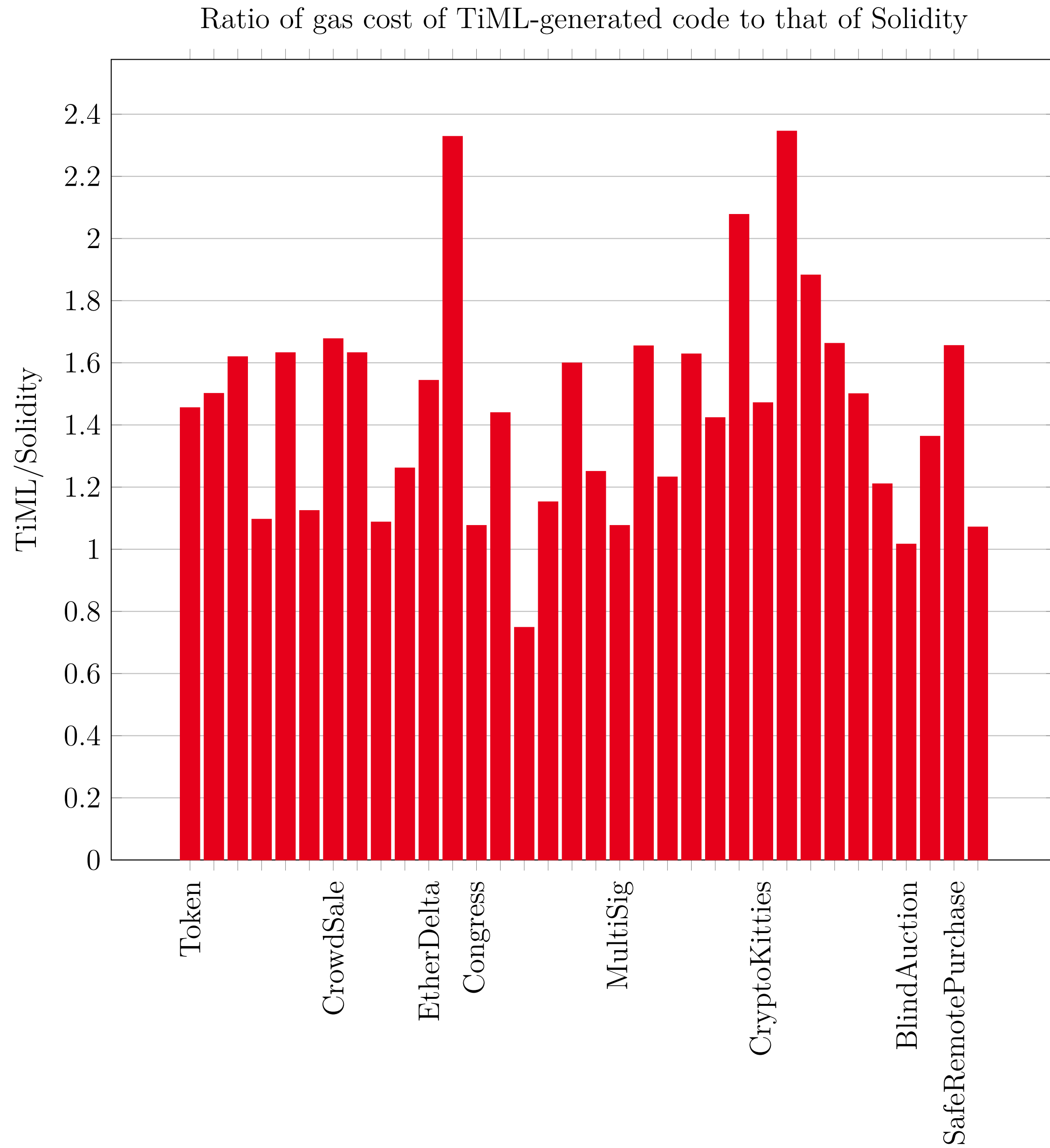

Figure 5-5: Same as Figure 5-4, except that the two functions with the largest slow-downs have been removed.

in TiML. TiML does not have built-in for-loops in the language but provides a higher-order function (or "combinator") `for` in the standard library that is similar to `foldl`[5]. Creating and calling functions are still relatively expensive in the current TiML implementation which involve allocating tuples on the heap to store free variables for a closure and live variables after the function call. I can measure the overhead from the for-loop (the `for` combinator) by running the loop with an empty loop body and then removing the loop altogether. After removing the for-loop overhead, `averageGen0SalePrice` and `tokensOfOwner` have slow-downs of 1.972 and 3.67 respectively. Barring these two functions, the slow-downs of all the other functions are illustrated in Figure 5-5.

As a minor data point, I also measured the time it takes to typecheck and compile each smart contract (with and without VC checking by an SMT solver), shown in Table 5.5 and Figure 5-6. The times are measured on a desktop machine with a quad-core Intel Core i7 7700K CPU at 5GHz.

Contract	Typechecking time (s)	Compilation time (s)	Typechecking time w.o. VC checking	Compilation time w.o. VC checking	LoC
Token	0.005	1.755	0.004	0.184	70
CrowdSale	0.004	0.485	0.003	0.057	92
EtherDelta	0.102	4.903	0.022	0.367	241
Congress	0.064	11.381	0.025	2.033	209
MultiSig	0.208	20.802	0.212	8.818	278
CryptoKitties	0.555	65.766	0.303	31.837	978
BlindAuction	0.312	8.253	0.007	1.009	98
SafeRemotePurchase	0.039	7.759	0.006	0.878	82

Table 5.5: Typechecking and compilation time. Typechecking time is included in compilation time. The fourth and fifth column show the times with the VC checking turned off. The last column shows the number of lines of code for each contract. The times are measured on a desktop machine with an Intel i7 7700K CPU at 5GHz.

I measured the typechecking time (included in the compilation time) because the programmer's attention is only required during typechecking (to prepare for error messages). After typechecking, the compilation is guaranteed to succeed. Fast typechecking provides a fast feedback loop for the programmer to debug compile-time

[5] `for` is a recursor for indexed natural numbers in the same way that `foldl` is a recursor for lists.

errors. TiML typechecking is fast on all the benchmark smart contracts, taking no more than 0.6 seconds, and I have not observed any situation where the typechecker takes a long time to report an error[6].

Compilation time is much longer than typechecking time because there are many typechecking phases for intermediate programs during the compilation, and these intermediate programs are significantly larger than the source program because of the inlining of index and type definitions. In a future version, index and type definitions will be supported by μTiML to avoid the inlining. And modulo bugs in the compiler, it is safe to skip the intermediate typechecking phases[7]. I also measured the typechecking/compilation time when the VC checking is skipped, to see how much time is spent by the SMT solver.

[6]Many systems suffer from the phenomenon that they finish quickly if the input is good but become very slow when the input has errors. TiML does not have the problem because (1) the typechecking algorithm's worst case is when the program typechecks, in which case the typechecker traverses the entire AST; (2) the VCs generated by the typechecker are simple enough that the SMT solver does not exhibit any asymmetric behavior.

[7]Annotations needed by the compilation phases can be provided by the first typechecking phase.

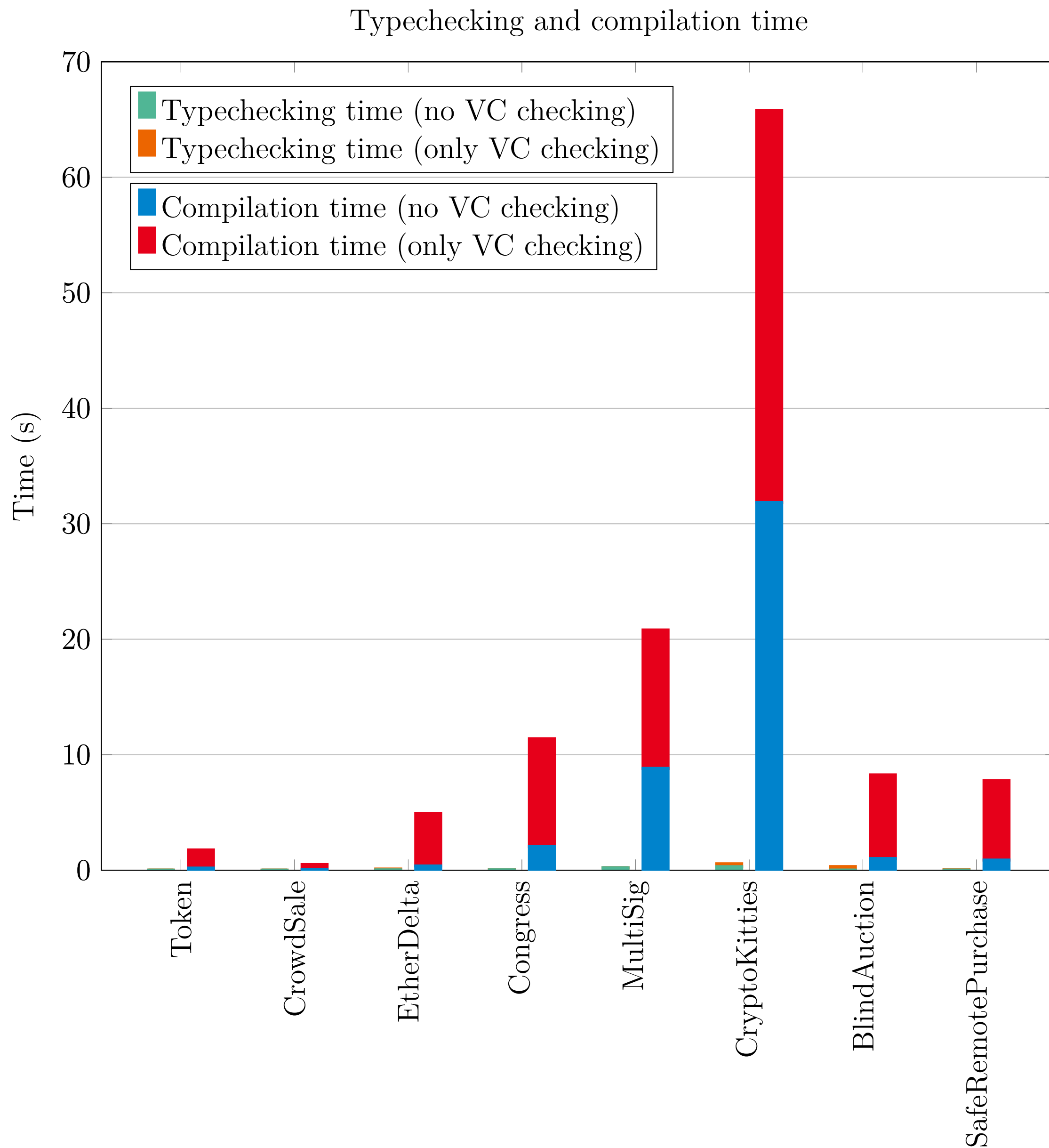

Figure 5-6: Typechecking and compilation time. This graphs shows the typechecking time (on the left, barely visible) and the total compilation time (on the right, including typechecking time), for each contract. The lower part in each bar represents the time with VC checking turned off, and the whole bar represents the time with VC checking turned on. Thus the upper part of each bar represents the time spent by the SMT solver used for VC checking.

Chapter 6

Related Work

Static resource analysis has been under study for about two decades and recently has gained more attention because of both technical breakthroughs and its potential value in software quality assurance and security [56, 29]. [43] called for "the great synergy" between the programming-languages community and algorithms community to eliminate the unfortunate gap between the study of the structure and efficiency of programs.

6.1 Dependent ML

The design of TiML is highly influenced by the Dependent ML (DML) language of [80]. The ideas of indexed types and refinement kinds are from their DML work. They did not explore the possibility of using their techniques to tackle the problem of static guarantees of program execution time. A follow-on project [38] also missed TiML's central idea that arrow types can be indexed by the function's cost bound. The work of [79, 80] has shown that their approach is powerful enough to accomplish tasks like static array-bound checking and dead-code elimination. I take these abilities for granted and only focus on resource-related capabilities in this thesis.

6.2 AARA and RAML

The Automatic Amortized Resource Analysis (AARA) line of work was initiated by [50] and successfully pursued by Hoffmann et al. The original idea was to associate a uniform potential with each list node in an affine type system. Aliasing is allowed by partitioning the potential among the aliases. Typechecking generates linear inequalities involving these (yet-unknown) potential coefficients, and solving this linear constraint system by a linear-programming solver can give a consistent assignment to these potential coefficients. The power of this idea is that it bypasses the difficult problem of recurrence solving altogether, yielding a push-button approach, at the cost of only supporting linear bounds on monomorphic first-order programs. Later work extended it to univariate [48] and multivariate [46] polynomial bounds by associating a uniform potential to not only one node but every tuple of nodes (of certain types); other extensions support higher-order programs [54], parallel programs [49], and a large portion of OCaml [47]. The bounds must be polynomial, and invariants are not supported. The latest treatment of higher-order functions [47] is somewhat unsatisfactory in that a higher-order function does not have a type that fully abstracts its behavior, meaning that at call sites the callee's code (not only its type) must be available to do typechecking and resource analysis. As a consequence, it is not possible to do separate typechecking/compilation where library functions' source code is unavailable. In contrast, TiML's typechecking is fully modular in that at a function's call site only its type is needed. The AARA approach also suffers from a common drawback among fully automatic tools that it disallows user-provided hints and help when automation fails.

The AARA amortization scheme is akin to "the banker's method" [74] and TiML's to "the physicist's method."

6.3 Program logics and verification systems

Program logics are usually good fits for reasoning about low-level programs. Using program logics on lower levels of a stack of formal systems that has a type system on its surface level also seems like a good idea. [16] introduced a program logic on JVM bytecode for resource verification formalized in Isabelle/HOL [1]. It is a partial-correctness program logic enjoying soundness and relative completeness, complemented by a termination logic. Parameterized on a resource algebra, the logic is very general, but verification (i.e. proving) is all manual, and resource recurrences (arising from the Consequence rule) must be solved by hand.

It is intended as a target language compiled from the type system of [50] in a proof-carrying-code [64] setting. This is the only other resource-analysis system I know of that involves a compiler. Unlike TiML's type-preserving compilation strategy, for the proposed compiler here, the source language is equipped with a type system while the intermediate and target languages are equipped with program logics. Unlike TiML, reconstructing proofs at the intermediate and target-language levels are not automatic. Actually the whole system is supposed to work within an interactive theorem prover, and the translation functions and their properties will be used as lemmas to prove the resource costs of a target program manually (given the typing derivation of the source program). I cannot find more information about this proposed compiler system, and it is not clear whether such a system has been built.

[17] proposed another program logic based on separation logic with resource potentials, together with a VC generator (requiring loop-invariant annotations) and a proof-search system that can automatically discharge VCs and use a linear-programming solver to infer resource annotations (influenced by [50]). This system, formalized in Coq, is close to TiML's design goal of supporting both highly automatic analysis and expressive invariants, though it is only for a first-order language. Designing program logics for higher-order languages is generally a hard problem.

[25] proved the inverse-Ackermann bound of union-find in Coq using their characteristic formulae (CFML) framework augmented with potentials. The proof is man-

ual, but the CFML framework (essentially an axiomatic semantics for OCaml) is shown to be very expressive. [62] introduced another Coq framework for time verification, which is based on a Coq monad that is indexed not by time but by a predicate mentioning time (similar to the Dijkstra monad of [73]) which can serve as both time and correctness invariant. Proofs are written manually. Both systems require "pay" or "tick" primitives to be inserted at time-consumption sites, either manually or by a program transformation.

[31] described an Agda library whose core constituent is a graded monad called "Thunk" whose index stands for the running time of this thunk. The novelty is the treatment of lazy evaluation with memoization, manifested by the "pay" primitive and the operational semantics for the thunked language. On the practical side, it suffers from the usual nuisance of working in an intensional dependent type system: indices must be definitionally equal for two types to be unifiable.

6.4 Sized types and refinement types

Sized types [69, 52, 75] are similar to indexed types in my setting, though the latter do not have any built-in size-related meaning while [52] gave a denotational semantics to the former relating types to their sizes. [75] described an algorithm for automatically generating cost recurrences from sized-type programs (but did not deal with recurrence solving). Sized types do not support refinements.

Çiçek et al. conducted a line of work [26, 24, 23] on analyzing incremental and relational time complexity. They use the term "refinement types" in the broad sense of "types enriched with other information," in contrast with my use in the narrow sense of "types of the form $\{x : t \mid P(x)\}$". They do not support restraining a type with an arbitrary predicate. Because the asynchronous rules in their type systems arbitrarily make the choice of relating which subpart of the first program to which subpart of the second program, the authors were unable to devise a typechecking algorithm.

[28] presented a type system for resource-bound certification with indexed types.

Indices and types are unified in their language, and inductive kinds and primitive recursion on the type level are supported. The program is intended to be machine-generated, so annotation burden is heavy (e.g. one cannot relax a time bound without annotating the "padding" amount). Refinements are not supported.

Liquid types [70, 77, 76] popularized refinement types as a practical middle ground between traditional ML types and full dependent types. The appeal of liquid types lies in their support of automatic refinement inference, made possible by fixing a set of qualifiers and iteratively weakening unsatisfied VCs by removing offending qualifiers. I would like to apply liquid-type techniques to enable automatic refinement inference in TiML, though these techniques are not very good at inferring constants in numerical formulas compared to e.g. counterexample-guided approaches.

6.5 Program analysis

Aside from the community working on type systems and software verification, complexity analysis has been studied for many years by the program-analysis community. The COSTA line of work [10, 12, 11, 13] aimed at cost analysis of Java bytecode. [11] made inroads in recurrence solving by converting cost relations to direct recursions and analyzing their evaluation trees. The latter is done by bounding the branching factors, the tree depths, and the sizes of the nodes. The SPEED line of work [39, 40, 41] did automatic complexity analysis of first-order imperative programs by instrumenting the program with multiple counters and using off-the-shelf abstract-interpretation-based linear invariant-generation tools to infer invariants on these counters automatically. [22] did complexity analysis of integer programs by alternating time and size analysis on small parts of the program. [71] handled non-linear theories by lazily instantiating theorems that are sufficient to approximate a nonlinear theory.

6.6 Gas analysis for Ethereum

[36] built a program-analysis tool working on raw EVM bytecode to detect vulnerabilities resulting from out-of-gas conditions. It does not give gas bounds or estimations, but detects code patterns that are likely to cause out-of-gas vulnerabilities, for which the paper gives some examples. [21] is a short paper where the authors translate Solidity code into F* [72] and verify program properties including gas upper-bounds. [14] tries to verify smart contracts in Isabelle/HOL. [45] and [37] aim to formally define the semantics of EVM.

6.7 Other resource-analysis systems

[59] studied complexity analysis of a first-order language where they focused on inferring constants in postcondition templates that can mention time, which is represented by an instrumented counter variable. Other than postconditions, it does not allow refinements in other places, so I do not see how it can, for example, relate a red-black tree's black-height to its size, which requires invariants of the data structure. [58] verified resource usage for higher-order functions with memoization by transforming the source program into a first-order program instrumented by resources and then generating VCs in Hoare-logic style. They also support index inference by counterexample-guided search. The idea of defunctionalization is also exploited by [18]. I do not want to use defunctionalization because I want to do fully modular complexity analysis, which requires analyzing higher-order functions without knowing the set of all possible argument functions. I can learn from [59] and [58] when it comes to index-inference techniques.

[35] introduced a resource-annotated operational semantics and type system for interaction nets, with the novel notions of "space-time complexity" and "scheduled types" for guaranteeing the availability pace of data. Since the interaction-net language lacks recursion, the programmer (not the language designer) has to define a new node for each operation like map/fold and its potential function. The paper

gives potential functions for the nodes it uses but does not show how to choose such potential functions for new operations. Annotation inference is not addressed, and invariants are not supported.

[30] did complexity analysis with linear dependent types, which are indexed linear types with a special index i meaning "the ith copy of this term." Linear dependent types enjoy relative completeness. [34] analyzed the complexity of a concurrent Algol-like language that is synthesized to hardware circuits directly, by using indices in types to control contraction in parallel compositions. [32] proposed a procedure to automatically transform a program into a certain form and read off the complexity-recurrence equations from there, but they did not address recurrence solving. [19, 20] used the Mathematica computer-algebra system to solve some forms of recurrences.

Appendix A

Technical details for TiEVM

A.1 TiEVM instructions (full list)

The following are the “real instructions”, those that come directly from EVM.

Add Mul Sub Div SDiv Mod
SMod Exp Lt Gt SLt SGt
Eq IsZero And Or Xor Not
Byte Sha3 Origin Address Balance Caller
CallValue CallDataSize GasPrice CoinBase TimeStamp
Difficulty CodeSize GasLimit Number MLoad MStore
MStore8 SLoad SStore JumpI JumpDest Push_n w
Dup_n Swap_n Log_n Pop CallDataLoad
CallDataCopy Jump Return

The following are a special kind of pseudo-instructions called “noop instructions” that expand to empty lists of real instructions.

$\text{Pack}^{\tau}_{\tau}$ PackI^{τ}_{i} Fold_{τ} AscType_{τ} Unpack_{α} UnpackI_{a}
Unfold AppT_{τ} AppI_{i} Nat2Int Int2Nat Byte2Int
AscTime_{i} AscSpace_{i} ArrayDone TupleDone RestrictView

The other pseudo-instructions are listed in Appendix A.2 along with their expansions.

A.2 Expansions of TiEVM pseudo-instructions

The expansions of TiEVM pseudo-instructions into real instructions are listed below. I use semicolons to separate instructions.

$$
\begin{aligned}
[\![\textsc{InitFreePtr}_n]\!] &= \textsc{Push}\ 32 \times n;\ \textsc{Push}\ 0;\ \textsc{MStore} \\
[\![\textsc{TupleMalloc}_{\overrightarrow{\tau}}]\!] &= \textsc{Push}\ 0;\ \textsc{MLoad};\ \textsc{Dup}_1;\ \textsc{Push}\ 32 \times |\overrightarrow{\tau}|;\ \textsc{Add};\ \textsc{Push}\ 0; \\
&\quad\ \textsc{MStore} \\
[\![\textsc{TupleInit}]\!] &= \textsc{Dup}_2;\ \textsc{MStore} \\
[\![\textsc{Printc}]\!] &= \textsc{Push}\ 32;\ \textsc{MStore};\ \textsc{Push}\ 1;\ \textsc{Push}\ 32;\ \textsc{Push}\ 31;\ \textsc{Add}; \\
&\quad\ \textsc{Log}_0;\ \textsc{Push}\ () \\
[\![\textsc{ArrayMalloc}_{\tau}^{n,b}]\!] &= \textsc{Push}\ 0;\ \textsc{MLoad};\ \textsc{Push}\ 32;\ \textsc{Add};\ \textsc{Dup}_1;\ \textsc{Swap}_2;\ \textsc{Push}\ n; \\
&\quad\ \textsc{Mul};\ \textsc{Add};\ \textsc{Push}\ 0;\ \textsc{MStore} \\
[\![\textsc{ArrayInit}_{32}]\!] &= \textsc{Dup}_3;\ \textsc{Dup}_3;\ \textsc{Dup}_3;\ \textsc{Add};\ \textsc{MStore} \\
[\![\textsc{ArrayInit}_{1}]\!] &= \textsc{Dup}_3;\ \textsc{Dup}_3;\ \textsc{Dup}_3;\ \textsc{Add};\ \textsc{MStore8} \\
[\![\textsc{ArrayInitLen}]\!] &= \textsc{Dup}_2;\ \textsc{Push}\ 32;\ \textsc{Swap}_1;\ \textsc{Sub};\ \textsc{MStore} \\
[\![\textsc{Int2Byte}]\!] &= \textsc{Push}\ 31;\ \textsc{Byte} \\
[\![\textsc{Int2Bool}]\!] &= \textsc{Push}\ 31;\ \textsc{Byte} \\
[\![\textsc{Inj}_{\tau}]\!] &= [\![\textsc{TupleMalloc}_{[\mathsf{unit},\mathsf{unit}]}]\!];\ \textsc{Swap}_1;\ \textsc{Dup}_2;\ \textsc{MStore};\ \textsc{Swap}_1; \\
&\quad\ \textsc{Dup}_2;\ \textsc{Push}\ 32;\ \textsc{Add};\ \textsc{MStore} \\
[\![\textsc{BrSum}]\!] &= \textsc{Dup}_2;\ \textsc{MLoad};\ \textsc{Swap}_1;\ \textsc{JumpI} \\
[\![\textsc{MapPtr}]\!] &= \textsc{Push}\ 32;\ \textsc{MStore};\ \textsc{Push}\ 64;\ \textsc{MStore};\ \textsc{Push}\ 64;\ \textsc{Push}\ 32; \\
&\quad\ \textsc{Sha3} \\
[\![\textsc{VectorPtr}]\!] &= \textsc{Push}\ 32;\ \textsc{MStore};\ \textsc{Push}\ 32;\ \textsc{Push}\ 32;\ \textsc{Sha3};\ \textsc{Add} \\
[\![\textsc{VectorPushBack}]\!] &= \textsc{Dup}_2;\ \textsc{Dup}_1;\ \textsc{SLoad};\ \textsc{Swap}_1;\ \textsc{Dup}_2;\ \textsc{Push}\ 1;\ \textsc{Add};\ \textsc{Swap}_1; \\
&\quad\ \textsc{SStore};\ \textsc{Swap}_1;\ \textsc{Swap2};\ [\![\textsc{VectorPtr}]\!];\ \textsc{SStore} \\
[\![\textsc{Halt}_{\tau}]\!] &= \textsc{Push}\ 32;\ \textsc{Swap}_1;\ \textsc{Dup}_2;\ \textsc{MStore};\ \textsc{Push}\ 32;\ \textsc{Swap}_1;\ \textsc{Return}
\end{aligned}
$$

Appendix B

Technical details for the compiler

B.1 Cost definitions

The cost parameters in TiEVM typing rules (see Section 3.4) are the same as those defined in the official EVM specification [5]. For a pseudo-instruction, its cost is simply the sum of the instruction costs in its expansion. Technically each cost parameter is a pair, and the second component (the memory cost) is always zero. The only exceptions are $C_{\textsc{TupleMalloc}}(n)$, whose second component is $32 \times n$[1], and $C_{\textsc{ArrayMalloc}}(i, n)$, whose second component is $n \times i + 32$.

The cost parameters in TiML typing rules (see Section 2.3.1) are defined below. Only those that do not differ across different compilation stages are listed here; the

[1]Since the expansion of Inj uses TupleMalloc to allocate a pair, the second component of $C_{\textsc{Inj}}$ is 64.

others are defined in Section 4.5.

$$
\begin{aligned}
C_{\mathsf{Const}} &= C_{\textsc{Push}} + C_{\mathsf{Let}} \\
C_{\mathsf{Tuple}}(n) &= n \times C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\textsc{TupleMalloc}} + C_{\textsc{Push}} + C_{\textsc{Add}} + n \times (C_{\textsc{Push}} + \\
&\quad 2 \times C_{\textsc{Swap}} + C_{\textsc{Sub}} + C_{\textsc{TupleInit}}) \\
C_{\mathsf{Proj}} &= C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\textsc{Push}} + C_{\textsc{Add}} + C_{\textsc{MLoad}} \\
C_{\mathsf{Inj}} &= C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\textsc{Push}} + C_{\textsc{Inj}} \\
C_{\mathsf{Fold}} &= C_{\mathsf{Var}} + C_{\mathsf{Let}} \\
C_{\mathsf{Unfold}} &= C_{\mathsf{Var}} + C_{\mathsf{Let}} \\
C_{\mathsf{Pack}} &= C_{\mathsf{Var}} + C_{\mathsf{Let}} \\
C_{\mathsf{Unpack}} &= C_{\mathsf{Var}} + C_{\mathsf{Let}} \\
C_{\mathsf{NatPlus}} &= 2 \times C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\textsc{Add}} \\
C_{\mathsf{Read}}(32) &= 2 \times C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\mathsf{ArrayPtr}} + C_{\textsc{MLoad}} \\
C_{\mathsf{Read}}(1) &= 2 \times C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\textsc{Add}} + C_{\textsc{Push}} + C_{\textsc{Swap}} + C_{\textsc{Sub}} + C_{\textsc{MLoad}} + C_{\textsc{Int2Byte}} \\
C_{\mathsf{Write}}(32) &= 3 \times C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\mathsf{ArrayPtr}} + 2 \times C_{\textsc{Swap}} + C_{\textsc{MStore}} + C_{\textsc{Push}} \\
C_{\mathsf{Write}}(1) &= 3 \times C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\textsc{Swap}} + C_{\textsc{Add}} + C_{\textsc{MStore8}} + C_{\textsc{Push}} \\
C_{\mathsf{ArrayPtr}} &= C_{\textsc{Push}} + C_{\textsc{Mul}} + C_{\textsc{Add}} \\
C_{\mathsf{Len}} &= C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\textsc{Push}} + C_{\textsc{Swap}} + C_{\textsc{Sub}} + C_{\textsc{MLoad}} \\
C_{\mathsf{ArrayFromList}}(m, n) &= n \times C_{\mathsf{Var}} + C_{\mathsf{Let}} + C_{\textsc{Push}} + C_{\textsc{Dup}} + C_{\textsc{ArrayMalloc}} + C_{\textsc{Swap}} + \\
&\quad C_{\textsc{ArrayInitLen}} + C_{\textsc{Push}} + n \times (2 \times C_{\textsc{Swap}} + C_{\textsc{ArrayInit}}(m) + C_{\textsc{Swap}} + \\
&\quad C_{\textsc{Pop}} + C_{\textsc{Swap}} + C_{\textsc{Push}} + C_{\textsc{Add}}) + C_{\textsc{Pop}} \\
C_{\mathsf{New}}(n, i) &= C_{\mathsf{NewLoop}}(n) \times i + C_{\mathsf{NewPreLoop}} + C_{\mathsf{NewLoopTest}} + C_{\mathsf{NewPostLoop}} + 2 \times C_{\mathsf{Var}} \\
C_{\mathsf{NewLoop}}(n) &= C_{\mathsf{NewLoopTest}} + 2 \times C_{\textsc{Push}} + C_{\textsc{Pop}} + C_{\textsc{Swap}} + C_{\textsc{Sub}} + C_{\textsc{ArrayInit}}(n) + C_{\textsc{Jump}} \\
C_{\mathsf{NewPreLoop}} &= 2 \times C_{\textsc{Swap}} + 2 \times C_{\textsc{Dup}} + C_{\textsc{ArrayMalloc}} + C_{\textsc{ArrayInit}} + 2 \times C_{\textsc{Push}} + \\
&\quad C_{\textsc{Mul}} + C_{\textsc{Jump}} \\
C_{\mathsf{NewLoopTest}} &= C_{\textsc{JumpDest}} + 2 \times C_{\textsc{Push}} + C_{\textsc{Dup}} + C_{\textsc{IsZero}} + C_{\textsc{JumpI}} \\
C_{\mathsf{NewPostLoop}} &= C_{\textsc{JumpDest}} + 3 \times C_{\textsc{Pop}} + C_{\textsc{Swap}} + C_{\mathsf{SetReg}}
\end{aligned}
$$

B.2 CPS cost adjustments

$$
\begin{aligned}
C_{\mathsf{CpsAbs}_1}(e) &= C_{\mathsf{CallK}}(e) \\
C_{\mathsf{CpsAbs}_2}(e) &= C^{\mathsf{PreCC}}_{\mathsf{Abs}}(FV(e)) \\
C_{\mathsf{CpsApp}_1}(k) &= C_{\mathsf{CpsAdjust}_1}(k) \\
C_{\mathsf{CpsApp}_2}(k) &= C_{\mathsf{CpsAdjust}_2}(k) + C^{\mathsf{PreCPS}}_{\mathsf{App}} \\
C_{\mathsf{CpsAbsT}_1}(e) &= C_{\mathsf{CallK}}(e) \\
C_{\mathsf{CpsAbsT}_2}(e) &= C^{\mathsf{PreCC}}_{\mathsf{Abs}}(FV(e)) \\
C_{\mathsf{CpsAppT}_1}(k) &= C_{\mathsf{CpsAdjust}_1}(k) \\
C_{\mathsf{CpsAppT}_2}(k) &= C_{\mathsf{CpsAdjust}_2}(k) + C^{\mathsf{PreCC}}_{\mathsf{App}} \\
C_{\mathsf{CpsCase}}(e_1, k) &= C_{\mathsf{CpsAdjust}_1}(k) + C_{\mathsf{CallK}}(e_1) \\
C_{\mathsf{CpsCase}}(e_2, k) &= C_{\mathsf{CpsAdjust}_1}(k) + C_{\mathsf{CallK}}(e_2) \\
C_{\mathsf{CpsCase}_0}(k) &= C_{\mathsf{CpsAdjust}_2}(k) + C^{\mathsf{PreCodeGen}}_{\mathsf{Case}} \\
C_{\mathsf{CpsAdjust}_1}(k) &= \mathsf{isVar}(k)\ ?\ C^{\mathsf{PreCC}}_{\mathsf{AbsInner}}(FV(k)) : (0,0) \\
C_{\mathsf{CpsAdjust}_2}(k) &= \mathsf{isVar}(k)\ ?\ C^{\mathsf{PreCC}}_{\mathsf{Abs}}(FV(k)) : (0,0)
\end{aligned}
$$

$\mathsf{isVar}(e)$ is a function testing whether e is a variable.

Bibliography

[1] Isabelle/HOL.

[2] Standard ML.

[3] Bitcoin, 2018.

[4] Ethereum project, 2018.

[5] Ethereum yellow paper, 2018.

[6] Go Ethereum, 2018.

[7] The Go programming language, 2018.

[8] Solidity, 2018.

[9] Solidity ABI, 2018.

[10] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In *Proceedings of the 16th European Symposium on Programming*, ESOP 2007, pages 157–172. Springer-Verlag, 2007.

[11] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proceedings of the 15th International Symposium on Static Analysis*, SAS 2008, pages 221–237. Springer-Verlag, 2008.

[12] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Costa: Design and implementation of a cost and termination analyzer for Java bytecode. In *Formal Methods for Components and Objects: 6th International Symposium*, FMCO 2007, pages 113–132. Springer Berlin Heidelberg, 2008.

[13] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa Gil. Live heap space analysis for languages with garbage collection. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM 2009, pages 129–138. ACM, 2009.

[14] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 66–77, New York, NY, USA, 2018. ACM.

[15] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2007.

[16] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theoretical Computer Science*, 389(3):411–445, December 2007.

[17] Robert Atkey. Amortised resource analysis with separation logic. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP 2010, pages 85–103. Springer-Verlag, 2010.

[18] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 152–164. ACM, 2015.

[19] Ralph Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, January 2001.

[20] Ralph Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2):79–103, June 2004.

[21] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.

[22] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference*, TACAS 2014, pages 140–155. Springer Berlin Heidelberg, 2014.

[23] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 316–329. ACM, 2017.

[24] Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. A type theory for incremental computational complexity with control flow changes. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 132–145. ACM, 2016.

[25] Arthur Charguéraud and François Pottier. Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In *Interactive Theorem Proving: 6th International Conference*, ITP 2015, pages 137–153. Springer International Publishing, 2015.

[26] Ezgi Çiçek, Deepak Garg, and Umut Acar. Refinement types for incremental computational complexity. In *Programming Languages and Systems: 24th European Symposium on Programming*, ESOP 2015, pages 406–431. Springer Berlin Heidelberg, 2015.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

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

[29] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium - Volume 12*, SSYM 2003, pages 3–3. USENIX Association, 2003.

[30] Ugo Dal Lago and Barbara Petit. The geometry of types. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2013, pages 167–178. ACM, 2013.

[31] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2008, pages 133–144. ACM, 2008.

[32] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 140–151. ACM, 2015.

[33] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2008/ETAPS 2008, pages 337–340. Springer-Verlag, 2008.

[34] Dan R. Ghica and Alex Smith. Geometry of synthesis III: Resource management through type inference. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2011, pages 345–356. ACM, 2011.

[35] Stéphane Gimenez and Georg Moser. The complexity of interaction. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 243–255. ACM, 2016.

[36] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2018.

[37] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 243–269, Cham, 2018. Springer International Publishing.

[38] Bernd Grobauer. Cost recurrences for DML programs. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP 2001, pages 253–264. ACM, 2001.

[39] Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV 2008, pages 370–384. Springer-Verlag, 2008.

[40] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2009, pages 127–139. ACM, 2009.

[41] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2010, pages 292–304. ACM, 2010.

[42] Robert Harper. *Practical Foundations for Programming Languages*, chapter 13: Pattern Matching, page 93–101. Cambridge University Press, 1st edition, 2013.

[43] Robert Harper. Structure and efficiency of computer programs. 2014.

[44] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM.

[45] Yoichi Hirai. Defining the Ethereum virtual machine for interactive theorem provers. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 520–535, Cham, 2017. Springer International Publishing.

[46] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2011, pages 357–370. ACM, 2011.

[47] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 359–373. ACM, 2017.

[48] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP 2010, pages 287–306. Springer-Verlag, 2010.

[49] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In *Proceedings of the 24th European Symposium on Programming Languages and Systems*, ESOP 2015, pages 132–157. Springer-Verlag New York, Inc., 2015.

[50] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2003, pages 185–197. ACM, 2003.

[51] Rodney R. Howell. On asymptotic notation with multiple variables, technical report. 2008.

[52] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 1996, pages 410–423. ACM, 1996.

[53] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[54] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2010, pages 223–236. ACM, 2010.

[55] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM.

[56] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO 1996, pages 104–113. Springer-Verlag, 1996.

[57] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR 2010, pages 348–370. Springer-Verlag, 2010.

[58] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 330–343. ACM, 2017.

[59] Ravichandhran Madhavan and Viktor Kuncak. Symbolic resource bound inference for functional programs. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, CAV 2014, pages 762–778. Springer-Verlag New York, Inc., 2014.

[60] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[61] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, Washington, DC, USA, 2001. IEEE Computer Society.

[62] Jay McCarthy, Burke Fetscher, Max New, Daniel Feltey, and Robert Bruce Findler. A Coq library for internal verification of running-times. In *Functional and Logic Programming: 13th International Symposium*, FLOPS 2016, pages 144–162. Springer International Publishing, 2016.

[63] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[64] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 1997, pages 106–119. ACM, 1997.

[65] Chris Okasaki. Three algorithms on Braun trees. *Journal of Functional Programming*, 7(6):661–666, November 1997.

[66] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[67] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 369–378, New York, NY, USA, 2015. ACM.

[68] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[69] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP 1994, pages 65–78. ACM, 1994.

[70] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2008, pages 159–169. ACM, 2008.

[71] Akhilesh Srikanth, Burak Sahin, and William R. Harris. Complexity verification using guided theorem enumeration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 639–652. ACM, 2017.

[72] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 256–270. ACM, 2016.

[73] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013, pages 387–398. ACM, 2013.

[74] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[75] Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Proceedings of the 15th International Conference on Implementation of Functional Languages*, IFL 2003, pages 86–101. Springer-Verlag, 2004.

[76] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 48–61. ACM, 2015.

[77] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP 2013, pages 209–228. Springer-Verlag, 2013.

[78] Peng Wang, Di Wang, and Adam Chlipala. TiML: A functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA):79:1–79:26, October 2017.

[79] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI 1998, pages 249–257. ACM, 1998.

[80] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 1999, pages 214–227. ACM, 1999.