Sorbonne Université

École Doctorale Informatique, Télécommunications et Électronique (ED 130)

LIP6 (UMR 7606) – Équipe Algorithmes, Programes et Résolution

Reusable Static Resource Analyses for High-level Languages

by **Hector Suzanne**

Ph.D. thesis in Computer Science

Supervised by **Pr. Emmanuel Chailloux**

Presented and publicly defended on March 20th 2025 to the jury:

Giulio Manzonetto, Rapporteur (Professeur, *Université Paris Cité*)

Sylvain Conchon, Rapporteur (Professeur, *Université Paris Saclay*),

Antoine Miné, Examinateur (Professeur, *Sorbonne Université*),

Julia Lawall, Examinatrice (Directrice de recherche, *INRIA Paris*)

Guillaume Munch-Maccagnoni, Examinanteur (Chargé de recherche, *INRIA Nantes*)

Simão Melo de Sousa, Examinateur (Professor catedrático, *Universitade do Algarve*)

Emmanuel Chailloux, Directeur de thèse (Professeur émérite, *Sorbonne Université*)

à Dada

Remerciements

Ce manuscrit est l'aboutissement d'un travail commencé durant le second confinement *Covid* en France, et terminé quatre ans plus tard à la suite de deux déménagements, une évacuation d'immeuble et bien d'autres péripéties. Il n'aurait assurément pas survécu aux âpretés communes aux travaux de thèses sans l'aide et le soutien ininterrompu de celles et ceux qui m'entourent. Leur sollicitude est trop grande pour être correctement citées.

Emmanuel Chailloux, Mon directeur de thèse, a été la colonne porteuse de cette thèse que je n'ai souvent pas pu être. Nous nous sommes rencontrés alors que continuer mes études me paraissait impossible, j'ai eu la chance singulière d'avoir été pris sous son aile. Sa sagesse, son activité et son humanité resterons gravées dans ma mémoire comme dans celle de tant d'autres.

Mes professeurs passés ont su m'infecter de l'amour des mathématiques et de l'informatique. M. Rigaudière et M. Sadania m'ont fait découvrir la beauté qui s'y trouvait. M. Guédes et M. Salmon m'ont montré la rectitude et l'effort qui les rendent proprement intéressantes.

La famille est un creuset dont on ne sort pas. Merci à ma mère de m'avoir toujours encouragé, écouté, et rassuré. Merci à mon père de m'avoir toujours soutenu et d'avoir fait des marathons, rien n'étant alors proprement impossible. Merci à Raphaël d'avoir toujours pu me changer les idées et d'être d'une grandeur d'âme rare et magnifique. Merci à Cléophée d'apporter de la jeunesse dans une vie trop confortable dans son sillon, et de m'avoir permis d'être un grand frère.

Merci à Mamou et Dada dans leur beau jardin à Fondettes. Ma madeleine de Proust est une madeleine mangée un de ces mercredis chez eux, dans la même cuisine où je faisais avec mon grand-père des amusettes scientifiques, où j'ai soudé mes premiers circuits électroniques. Quelque chose de là-bas a germé et a donné des fruits.

Enfin, à toutes et tous les autres dont j'ai échoué à mettre sur la page les gentillesses qu'ils m'ont faites encore et encore, merci du fond du cœur.

Merci à tous,

Hector

Abstracts / Résumés

Short abstract

Programmers care about the amount of time, memory, and energy their program require to run. This is not only a matter of ecological ethic, but also of safety: software can fail or degrade in quality when not enough resources are available.

This thesis attempts to improve resource analyzers, the tools programmers use to evaluate those resource bounds past which their code fail. Many experiments and prototypes show those analyzers can exist, but they each tailored to a particular kind of software or kind of operations programs run. We would need those combined together to obtain useful resource bounds in general.

To this end, we introduce a new way to combine resource analyzers on top of a "neutral ground" that is compatible with many already-existing analysis, whereas previous works built each analysis on its own specific foundation, making them mutually incompatible.

Résumé court

Il est important pour les programmeuses et programmeurs de maîtriser la quantité de temps, de mémoire, et d'énergie que leurs programmes consomment. Il s'agit non seulement d'un enjeu écologique, mais aussi une question de sûreté: les logiciels peuvent échouer si les ressources dont ils ont besoin n'existent pas en quantité suffisante quand on les exécute.

Cette thèse cherche à améliorer la situation des analyseurs de ressources pour programmes, les outils qui permettent de calculer à l'avance les ressources nécessaires. Il existe de nombreuses expériences et prototypes qui montrent que ces analyses sont faisables, mais il faudrait pouvoir les combiner pour obtenir des résultats utiles.

Pour ce faire, nous introduisons une manière de combiner ces analyses de ressources dans un "terrain neutre" qui est compatible avec plusieurs analyses déjà développées, alors que précédemment, chaque analyse avait sont propre modèle incompatible avec les autres.

Abstract

Programmers care about the amount of time, memory, and energy their program require to run. This is not only a matter of ecological ethic, but also of safety: software can fail or degrade in quality when not enough resources are available. This work aims to improve static resource analyzers by proposing a neutral formalism capable of combining several analysis techniques and of reusing them of different languages. It is based on an abstract machine and corresponding type system through the Curry-Howard correspondence. At type-level, it embeds the intuitionistic linear sequent calculus together with "parameters" at the second order. Those parameters are manipulated with the conjunctive-implicative fragment of classical first-order logic, with a user-provided signature. At term level, this corresponds to a call-by-push-value virtual machine, with explicit resource manipulation. Parameters represent characteristic quantities of data structures and computations (sizes, iteration counts, etc.)

Our type system generates, together with a typed program, a first-order constraint over the parameters used in it. Those include both the free parameter variables, but also those bounds by quantifiers. This constraint being first-order, it is amendable to SMT solving. Furthermore, in the important case of integers with signature ($\mathbb{N}$, 0, 1, $+$, -, *), those quantifiers can be eliminated without significant loss of predictive power in cases found in the wild. This quantifier-free constraint can then be turned into an integer optimisation program, giving closed-form bounds on the values of those parameters. This provides resource bounds to programs written in the machine language. Our abstract machine admits a effect system capable of soundly encoding resource-passing without adding new primitives. This facilitates resource analysis for other languages: any compilation scheme to the machine automatically extends to a resource-aware one, which is sound regarding the original semantics.

We show the feasibility of our method through its implementation, "AutoBill". It takes as input either a ML-style call-by-value language, a Call-by-Push-Value lambda-calculus, or in machine languages. Parameters with arbitrary user-defined sorts are supported. AutoBill then generates the corresponding constraint in a standard format. The Z3 solver can then provide closed-form bounds on those parameters. The extensibility of our method is demonstrated through the addition of a monadic effect system in the ML-language, and through the encoding of AARA analyses. AutoBill supports parameters annotations for user-described size and complexity, which previous AARA implementations didn't support.

Résumé

Il est important pour les programmeuses et programmeurs de maîtriser la quantité de temps, de mémoire, et d'énergie que leurs programmes consomment. Il s'agit non seulement d'un enjeu écologique, mais aussi une question de sûreté : les logiciels peuvent échouer si les ressources dont ils ont besoin n'existent pas en quantité suffisante quand on les exécute.

Cette thèse cherche à améliorer la situation des analyseurs statiques de ressources, en proposant un formalisme "neutre" capable de combiner plusieurs analyses et de les réutiliser sur plusieurs langages de programmation. Ce formalisme est basé sur une machine abstraite et son système de type idoine via la correspondance de Curry-Howard. Au niveau des types, ce formalisme incarne le calcul des séquents linéaire intuitionniste, augmenté au second ordre par l'addition de "paramètres", des variables de type de sortes quelconques. Ces paramètres sont manipulés avec le fragment conjonction/implication de la logique classique du premier ordre, avec une signature fournie par l'utilisateur. Au niveau logiciel, ce système correspond à une machine virtuelle Call-by-Push-Value, avec gestion explicite des ressources, et les paramètres représentent des grandeurs caractéristiques des programmes (tailles, nombre de tour de boucle, etc.).

Le système de type de cette machine abstraite produit, en plus d'un programme typé, une contrainte du premier ordre sur les paramètres du programme, à la fois les paramètres libres et ceux liés par des quantificateurs. Cette contrainte étant exprimée dans un fragment de la logique du premier ordre, il est possible de tester sa validité ou sa satisfaisabilité avec un solveur SMT. De plus, dans le cas de paramètres dans signature (int,0,1,+,-,*), il est possible d'éliminer les quantificateurs sans réduire le pouvoir de prédiction dans les cas réalistes. La contrainte résultante peut alors servir de base pour un programme d'optimisation sur les entiers fournissant des bornes sous formes closes de paramètres. Cela permet de fournir des bornes de ressources pour les programmes décrits dans le langage de la machine.

La machine abstraite que nous présentons admet de plus un système d'effet de bord permettant d'encoder le passage de ressource dans les programme sans besoin de primitives supplémentaires. Cela facilite la compilation de langages sources dans la machine : un schéma de compilation sans ressources engendre automatiquement un schéma avec ressources, tout en respectant la sémantique opérationnelle du langage source telle que décrite par le premier schéma.

Nous démontrons la validité de cette approche via une implémentation "AutoBill". Celle-ci peut prendre en entrée des programmes écrit dans un langage à la ML avec sémantique en appel par valeur pour en Call-by-push-value, ou en langage machine. Ces programmes peuvent inclure des paramètres et des signatures arbitraires. AutoBill renvoie alors la contrainte correspondante dans un format de fichier standard. Le solveur Z3 peut alors fournir les formes closes voulues. Nous présentons l'extensibilité de notre approche en ajoutant des effets de bord monadiques au langage à la ML, et en encodant l'analyse de complexité AARA dans les programmes. Dans ce dernier cas, AutoBill supporte les annotations de tailles et complexités, ce que les implémentations précédentes d'AARA ne supportaient pas.

TABLE OF CONTENTS

Chapter 1

Introduction

How can one predict the hardware footprint of a program? What compromises must be made to have such predictions? Programming is often a significant investment of efforts and time, and software mishandling its time, storage, or energy can be a disheartening surprise, if not a recurrent struggle. Programs are *material* entities as much as *abstractions*. While this is a truism, it has far-reaching consequences worth mulling over. Programs obviously have a footprint in the "real world", as they run on computers. This determines, for example, the time and energy footprints of programs in an obvious manner: it can be measured by an external observer.

Machines alone do not allow for resource predictions: how much resource a machine could ever possibly consume is not the matter at hand here. We want to know what resources are needed to run a particular program. Programs are made of code, which is where their synthetic, abstract nature shows. As such, predicting program behavior in general – and resource footprint in particular – needs to combine an *operational* viewpoint, in which a program is a behavior through time and space that can be measured, and a *structural* viewpoint, in which the patterns within code can be exploited.

Useful programs usually behave differently depending on their inputs, which means their footprint cannot be determined by merely "stepping through the code": we would then need to know in advance its input. There is a tension between the information we can glean from the code and the variability of inputs. Predicting the behavior of computers by means of analysis of their code is the concern of *software correctness and safety*. This discipline is fundamental importance given the scale and centrality software has today, and the effects of its potential

dysfunctions. Those disciplines exist at the frontier between the abstract, logical world of code and the operational, material world of machine, as we'll now see in more detail.

1.1 Code, behavior, and safety

There are two complementary means to ensure software safety: by proving that the code being run is correct, and/or by asserting that the machine running it behaves as expected. In general, software safety cannot be ensured by mere observations of the machine, as those do not suffice to characterize potential behaviors in all possible situations: given an input, the behavior of*one run of the program* against that input can be observed, but the behavior of a program itself cannot be tested *over all* potential inputs. Therefore, if one wants to predict the behavior of a program with maximal certainty, the only hope is to focus on its code. When it comes to resource analysis, this means that resource footprints should be derived by analysis the code of a program, giving a closed form of that footprint as a function of some salient features of its potential inputs.

Analyzing code – which remains unchanged over all runs of a program – is a sure-fire way to make correct predictions about program behavior. The question then becomes, to which extent do those predictions justify the so-called safety of the machine? For example, many programs – whose safety would otherwise be ensured by code analysis – fail when the memory they require isn't available on the particular machine that runs them. Code only gives an partial, implicit specification of the machine behavior. As such Even when code explicitly represent intended behavior, good predictions might be out of reach. Analyzable code must satisfyingly specify its intended behavior, and analysis systems must be sophisticated enough to extract relevant and actionable information from this. A tension exists between programmer efforts in writing code in a way that can be analysed, and the efforts of analysis designers in writing sufficiently sophisticated analyses to explicit the behavior of code in as many situations as possible. There are many possible avenues for such analysis, with diverse requirements, precision, and ease-of-use for programmers. In this endeavor, the choice of programming language is supremely important, as it serve as the interface between programmers and analysers. It dictates what is explicit and implicit within programs.

1.2 The Curry-Howard correspondence

Programming hasn't always involved controlling computers with code. The first mechanical and electronic computing devices were programmed by directly interfacing with their hardware (by setting pins or gears, wiring, and setting individual bits). The very term "compiler" wasn't introduced until 1953[81]. Since then, the face of programming has changed, thanks to the freedom afforded to programmers from programming as the machine demanded. This allowed for higher productivity through the elimination of irrelevant work, and higher safety through both automated code generation, and checks and balances implemented inside compilers which can detect predetermined classes of programmer errors. The resulting increase in abstraction led to the multiplication and success of *high-level* programming languages.

This bottom-up account of programming should be paired with another top-down account of abstraction in programming. From the latter perspective, concepts and techniques from formal logic and symbolic reasoning were applied to computer programming and software engineering. Starting with McCarty's LISP, whose function-denoting `LAMBDA`, acting through syntactic substitution,was found to be a direct mimicry of λ-calculus, formalisms from formal logic were linked to programming tools, forming a fruitful bridge between disciplines centered around the *Curry-Howard correspondence* (CHC). This correspondence is made of a vast network of links between formal models of code, processes and logical reasoning. That is to say, processes evolving in computers are linked to rewriting of their code, which can be mapped to reasoning on their *types*, to derive in turn properties of the evolution of those very processes.

For example, a computer with a CPU, a call stack, and `call`/`ret` instructions can be programmed with λ-terms. At type level, the abstraction $\lambda x.e$ is given the function type $A \Rightarrow B$, its argument e' the type A, and the overall application $(\lambda x.e)(e')$ the type B. There is then a formal correspondence between, on the one hand, the transitions between the states before `call` is executed and `ret` is, on the other hand, the β-reduction in the λ-calculus, and on the third hand the rule of *modus ponens* that turns A and $A \Rightarrow B$ into B. This particular correspondence between functions, implication, and stack machines is ubiquitous in the style of programming using the correspondence to build programs in a principled way, namely *functional programming*.

1.3 Functional programming for safety and correctness

The CHC covers a wide portfolio of qualitative and quantitative machine behavior, programming languages, and logical systems. The resulting construction is a "Rosetta Stone" of prediction, that provides an ever-expanding corpus of mutually-compatible tools that link predictions about machine behaviors with symbolic reasoning and rewriting theory. This directly enables the use of powerful techniques from mathematics and logic to help programmers understand and justify the programs they write. Programming languages and accompanying tooling can enable a higher level of correctness and safety from the use of the CHC and high-level programming. To quote a folklore slogan: "Well-typed programs do not go wrong"[1].

For example, functional programs written in any of the member of the ML family of languages (OCaml, SML, Haskell, ...) are guaranteed to be free from invalid memory access, as opposed to programs written in C/++, Java, or Python that provide nullable types, which can contain undefined references (`nullptr`, `null`, and `None` respectively). It would be fairer to say that well-typed programs do not go wrong *in the ways the compiler checks against.*

Functional programming also exhibits the inherent partiality of code analysis. For example, OCaml exceptions are not tracked by the type system, and OCaml programs may terminate unexpectedly and undesirably due to an uncaught exception, unless separate tooling is used to alleviate this particular issue. On the other hand, Haskell's monadic exceptions are tracked by the compiler and are always coupled with a corresponding handler, preventing the risk of failure[2]. If one wants to guarantee some correctness property of programs, one must type for it. Since we want to predict resource usage of functional programs, we will build a resource-aware type system capable of quantitative reasoning.

1.4 Memory safety for FP

Memory usage is an important concern for programmers, but guarantees in that aspect are often not provided for in tooling of the ML-family. Indeed, the automatic memory management necessary for their structural safety completely elides memory management in source code. For

[1] The aphorism is from Milner, of Hindley-Milner type inference fame, and recorded in [60]

[2] Haskell also possesses `IOExceptions` which aren't type-checked and may cause program failure, notably used to define non-total functions in the standard library like `head :: [a] -> a`

this reason, programs having stringent memory requirement or requiring multi-model memory management are written in programming languages with manual memory management, using its explicit nature to limit allocations to some specified bounds, and provide a *lingua franca* between paradigms. In general, the entire topic of memory management in programs remains a non-trivial problem, and many schools and techniques exists across languages, frameworks, and projects.

This motivates the creation of general theoretical frameworks for memory analysis and their implementations, to serve as laboratory benches to explore the design space it encompasses. Furthermore, statically approximating or determining memory management is, a *canonical* resource prediction problem. Indeed, memory is a recoupable resource: as opposed to time of energy, memory can be recovered when freed and then reused without incurring extra costs. It therefore subsumes resource prediction for non-recoupable resources. It also is minimal: the answer of a memory bounding problem is a single number or numerical expression (an integer numbers of bytes or blocks of memory) as opposed to more structured data.

While giving a comprehensive tour of those memory management techniques is beyond the scope of this introduction, some are of sufficiently large adoption and are embedded deep enough within programming practice as to deserve in our eyes closer attention:

Malloc The `malloc` function is the standard memory allocator for the C programming language since the seminal "K & R" book[47]. Memory allocated by `malloc` is guaranteed to be available until a call to another memory management function such as `free` for de-allocation or `realloc` for size changes. Mismanagement of those calls leads of a wide array of bugs such as *use-after-free* and *free-after-free*. Multithreaded programming brings new classes of pitfalls, for example race condition between *use* and *free* in different threads, which cause spurious and hard to reproduce crashes due to illegal memory access. Due to its inherent lack of safety guarantees, *malloc*-style memory management isn't widely used in functional programming languages.

RAII Use of the manual memory management paradigm can be made easier with techniques such as *Resource Acquisition Is Initialization* (RAII), in which memory is allocated only at the creation of objects, who all possess destructors responsible for freeing the memory those objects control, and calling other destructors when necessary. One significant

example of RAII providing safe, statically determined de-allocation, and to which we'll come back to in the next section, is the Rust programming language.

GC Garbage Collection is a technique allowing memory management to be safely automated. At the surface level, the programmer can dynamically allocate memory, and needs not to care to free or re-use it after the data it holds has become useless. Behind the scene, an automated garbage collector partitions memory allocations as "possibly accessible" or "provably out-of-scope", and frees the latter. This is done, starting from a set of *roots* containing all directly accessible data, by following references transitively over all accessible memory locations. This processing requires the program to be paused while memory is cleaned. The orchestration of allocations in memory and of GC pauses in time active topic of engineering and research. Garbage collection is used by OCaml, Haskell, and Standard ML of New Jersey, but also JavaScript and Scheme, making it the main memory management strategy in functional programming, both statically and dynamically typed, and is also used in languages implemented on top of the *Java Virtual Machine* (JVM), such as Java itself, Clojure, and Kotlin.

RC Reference counting is another approach to automated memory management in which allocation reachability is computed ahead of time by mean of a *reference counter* embedded in each allocation. Counters are incremented every time a new reference to their host allocation is created, and decremented whenever one of those references goes out of scope. When the counter reaches zero, the allocation can safely be freed. RC is used in programming platforms such as Microsoft's *Component Object Model*, Apple's *Cocoa* and *Swift*, and the *Python* reference implementation *CPython*. Note that RC and GC aren't mutually exclusive: A garbage collector may use reference counts to compute reachability, but only free memory during pauses, and reference counting often uses techniques coming from garbage collection to handle reference cycles in allocated data, which otherwise would keep counters from ever reaching zero. CPython, for example, can also be considered a GC implementation in this regard.

As this list suggest, the existence and variety of memory management paradigm causes the memory footprint of programs to depend on fine-grained design choices and on implementation details. This implies that, to figure out precisely the memory footprint of a program, an analysis must take into account a wide variety of concerns: programs behavior depends not

only on the size of data, but on its structure as well, which means simple numerical reasoning is often not enough to compute tight bounds. Just as well, if a loop allocates at each iteration, the total size of those allocations must necessarily depend on the number of iterations. When looking at memory prediction for functional programming from this angle, as a canonical example of static analysis of significant importance, it becomes natural to ask what can memory management and complexity analyses learn from one another.

1.5 Reasoning about memory, size and resources in FP

When it comes to memory management, most functional typed languages have an automated approach in which dynamic check are used to guarantee safety and attempt to use memory as best as possible. But the *Rust* programming language uses a different strategy altogether: all (de-)allocation points are statically determined from the source code of the programs. Following RAII, data structures have *ownership* of the memory they occupy, and when those structures go out of scope, that memory is freed. This is trivial if a unique reference to the data exists, but for shared data, memory liberation is prevented for the entire *lifetime* of the shared copy. To ensure this invariant is preserved across read- and write- accesses coming from different threads, Rust uses a *borrow checker*.

The technique of borrow checking formalizes RAII in a sophisticated type system that provides safe memory access*by construction* thanks to the CHC. It is a part of a line of programming languages that exploit *linear logic* for that purpose. Linear logic refines the usual classical and intuitionistic logics by restricting duplication and deletion of logical entities. The entities that cannot be duplicated or deleted are called *linear*, and enable a direct encoding of resources in logical reasoning (and by extension, in programs). In Rust, a slight relaxation is used, which allows the memory hosting a value to be automatically deleted when it is guaranteed no more references to that value exists. This may pose problems in the case of cyclic data structures, which require more work from the programmer. This can be done, for example, by representing the cyclic structure and the memory hosting it separately, at which point it is safe to free all the memory at once when no references to any part of the structure remains in scope.

Since duplication of references is forbidden, read/write access to a shared allocation require a *borrowing* for a certain duration. This duration is called the *lifetime* of the borrowed

reference, and lifetimes are partially order under inclusion, allowing the *borrow-checker* to guarantees that shared references cannot be misused, and that freeing allocations will not cause a *use-after-free*, *free-after-free*, or race conditions during concurrent read/writes. In general, this statically-determined ownership cannot be as precise and convenient as automated techniques with dynamically-determined memory management points. Continuing the example of a cyclic data structure, handling the entire memory extant of the structure as a whole prevents freeing and reusing of fragments of it, even when that reuse in possible.

RAII and linear logic can form the core of resource and memory analysers for functional languages, and are, for example, a core feature of the Rust programming language, which validate the feasibility of this method. But this does not suffice, as tools extracting computational costs and complexities must be added to this base to obtain memory footprints.

We now turn our attention to those tools. Analysis of high-level source code (i.e. irrelevant of machine consideration) was first done by hand on pseudocode for *algorithmic analysis*. The programming model used matters for such analysis: each language has different constructs for branching, iteration, looping, etc., which influence how programs can be written in theory and which program is written in practice. This in turns determines which methods can successfully analyse the complexity of programs. For example, Martin Hofmann proved in a seminal result[36] that algorithms implementable in polynomial time on a Turing-machine are exactly the functional programs on bit-strings implementable with high-order functions, structural recursion, and no dynamic memory allocation. Here, "no dynamic allocation" is understood in the sense that constructing data and closures consumes resource tokens that can only be created by freeing other data or closures of equivalent size. In this result, linear logic is directly used to represent preservation of a quantity of resources. Developments of resource analysis for functional languages using such techniques will be presented in the next chapter.

1.6 Thesis of this manuscript

There is a plethora of useful but not generally applicable resource analysis for ML-style languages based on the CHC. At their core, they use the correspondence between λ-calculus and minimal logic: predictions made on code remain true throughout execution because the reasoning that justify them are stable under reduction. Formally, reduction of typed λ-terms with resources matches with normalization of proofs in minimal linear logic. But looking at this

situation from the formal logic side, a possibility for improvement appears, as minimal linear logic is not the most straightforward incarnation of linear logic (an account of this phenomenon is given by Girard in [27]). A more efficient formalism to express linear logic is sequent calculus which trades the minimalism of implication logic (only one operator) for a maximalist formalism in which all operations possible on the state of a reasoning are given an orthogonal primitive operator. This formalism has been at the heart of the development of powerful techniques capable of analyzing their behavior of proofs in linear logic. Through the CHC, those become techniques to control and analyse the behavior of *abstract machines* using type systems, in the same way those type systems are used for λ-calculus. This enables implementing sophisticated type systems for functional languages while enjoying a comparatively low amount of proof work.

We therefore assert that resource analysis for functional typed languages benefit from a reformulation as type systems on a relevant abstract machine which we describe in this thesis, together with a reusable analysis scheme. We shall go into detail about this approach at the end of our description of the state of the art in chapter 2. To give a taste of this method, we describe some salient points:

- We describe the resource footprints of programs as the size of an ambient pool of resources used throughout the execution, to with all resource manipulation are logged. This implies resource bounds are preserved along evaluation. This would not be the case if the footprint was defined as the minimum amount of resources required for execution to proceed correctly, which would decrease as costly operations are done execution. This means that, the same way the type of a program does not change during evaluation, our resource footprints enjoy subject reduction, and will not change during evaluation.

- We shall represent this pool and log as a single linear value, making it obvious that resources are not spuriously created during evaluation. Also, as opposed to other current approaches. This dispenses with sophisticated techniques required to account for all resources spread around the program's state as it runs. Nevertheless, we do not lose precision compared to those resource analyses.

- Our machinery will be able to account for the precise operational semantics of source programming languages: as opposed to theoretical λ-calculus which needs to be wrangled with to match the order of operations in a real programming language, formalization uses

focusing to match the reality of step-by-step program evaluation. This match between theory and practice enables reuse of the machinery and analysis for new programming languages.

- Finally, we make explicit in our formalism the use of key software component present in implemented resource analyses, namely, *Solvers Modulo Theory* (SMT). All resource analyses require numerical reasoning on unknown quantities, such as the size of data structures, the number of iteration in a loop, and of course amount of resources. Our system makes this dependency explicit, is formalized using first-order logic, which we consider to be a neutral encoding. Not only is it directly understandable by SMT, It allows specific formalisms to be ported and combined more easily.

With this setup, we are able to analyse a functional intermediate representation with algebraic data types, recursion and amenities for monadic effects. Independently of this analysis, the machine code is automatically re-written to manipulate resources explicitly. This elaborated code can then be type-checked to compute a logical constraint on resources, which encodes an implicit complexity bound in terms of user-chosen quantities. This constraint is expressed in first-order logic, and can be sent to relevant solvers to obtain closed-form bounds or simplified constraints on memory requirements.

This setup has several advantages to previous tooling made for resource analysis for functional programming. It automatically mixes evaluation styles, both eager and lazy, within the same program. This is used to implement monadic effects and resource-efficient structural recursion schemes with iterators over user-defined data types. Also, to enable using a wide range of specific algorithmic analyses in this setting, our method extracts generic logical constraints more straightforwardly compatible with diverse analyses compared to made-to-purpose annotations, without changes to the compilation frontend or our typing procedure.

1.7 Outline of the argument

Our argument will begin with an overview of resource analysis techniques for functional programming in chapter 2. We begin with *Amortized analysis* in section 2.1, which we then formalize and re-frame in sections 2.2 and 2.3. Then, we describe the framework of *Automated Amortized Resource Analysis* (AARA, sec.2.4), its application to automated analysis for

functional languages *à la* ML (sec.2.5), and the algorithmic core of its inference system in section 2.6. We shall then describe a theoretical universal framework for hosting resource analysis called dℓPCFin section 2.7, and its direct application to amortized analysis in section 2.8. Finally, we introduce abstract interpretation in the context of resource analysis, and explain its complementary nature to AARA techniques (sec.2.9), before giving closing remarks on this review of the state of the art in section 2.11.

We will then start in earnest by presenting of the abstract machine-based approach that we will use, in chapter 3. We begin by restating the definition of call-by-value and call-by-name small-step operational semantics of λ-calculus in terms of choices of evaluation contexts in section 3.1, and reify those contexts to define a base abstract machine in section 3.2. From then on, we relate those abstract machines with *focusing* in logic, and show how it corresponds to evaluation strategies for abstract machines through the CHC (sec 3.3). We then use this correspondence to define call-by-value and call-by-name machines through minimal alterations to the base abstract machine in section 3.4 and 3.5. This allows us to define in section 3.6 an simultaneous embedding of λ-calculus into the two abstract machines, which endows it with call-by-value or call-by-name semantics depending on the target. We conclude with qualitative insights giving a first justification to our abstract-machine approach to resource analysis in section 3.7.

In chapter 4, we obtain an abstract machine combining the call-by-value and call-by-name machine by finalizing the application of focusing, the result being a*call-by-push-value* abstract machine. The base of the machine, its *identity fragment* is given in section 4.1, and we follow with its simple linear type system in section 4.2. Then, we proceed with the *logical fragment*, in which we introduce the definition of new types (sec. 4.3), the usage of inductive data-types (sec. 4.5), and of co-inductive computation types (sec. 4.6). We introduce thunks and closure to the machine, which mediate between evaluation strategies, in section 4.7, and comment and exemplify their use and relation to focusing in section 4.8. Finally, we introduce the *structural fragment* in section 4.9, which concerns itself with shared data and recursive computations. We conclude this presentation in section 4.10.

With this analysis in our toolbox, we show in chapter 5 how an ML-style functional programming language can be compiled into the machine. Finally, we demonstrate how the choice of abstract machine allows monadic effects to be taken into account without changing the

core analysis procedure. The whole forms the frontend of our analysis platform, implemented component-by-component on top of our reusable base.

Chapter 6 will then focus on the backend of the analysis, starting with its annotation system. This is formalized as a fragment of first-order logic encoding relations between resources, sizes, lengths, etc. used within the program, which we call *parameters*. This logical fragment is then used to extend our type system with polymorphism over parameters and parameterized simple types, and with the introduction of constraint within code. Well-typed programs then include a constraint within their typing judgment, free variables representing approximations of runtime metrics on manipulated data and computations.

We then introduce a representation of resources and potential using this system in chapter 7, which allows us to implement AARA as a type inference procedure. The logical constraint associated to the annotated program has then models which guarantee that enough resources are present at runtime, giving our footprint. We extend this translation with, on the one hand, annotations to datatype definitions, which allow us to encode a variety of resource analyses in the literature, and on the other with shared values with bounded lifetimes, which enables an analysis of memory-managed data structures.

In chapter 8, we describe our implementation of this analysis. Starting with a bird's eye view of our pipeline, we then spend time describe the mechanization of our type system into an inference procedure using constraints-based type inference procedures. We show how we extract a first-order logic constraint from our procedure, and provide initial solving and optimization algorithms to obtain bounds in the case of polynomial memory complexity. We then describe how our implementation interfaces with solvers and proof-assistants for external processing of constraints.

Finally, chapter 9 is dedicated to an evaluation of our work against the state of the art described in chapter 2. First, we show how our novel call-by-push-value machine enables an increase in the scope of analyzable programs. Then, we discuss the effect of allowing annotations in analysed code, especially in the context of higher-order programs, and the benefits of allowing state-of-the-art constraints solvers to be used in AARA analysers as opposed to standalone implementations of ad-hoc formalisms. We then discuss the relation of our continuation-passing, focused abstract machine with the *stack-aware* type systems used in state-of-the-art AARA systems, and open up a discussion about the possibility and difficulty

of re-using past AARA solving procedure in our new setting. We finally touch on also the opportunity a more standard first-order-logic formalism provides here. This will enable us to conclude our manuscript in chapter 10.

Chapter 2

Predicting resource consumption

When an program is functionally correct, its performance is a major concern for its authors and users. In the earlier days of computers, programs would statically assign different segments of memory for specific purposes, but the increase in complexity of programs and computing power of machines relegated this approach to niches within the wider programming practice. At it stands nowadays, and has been the case for decades, a programmer cannot be expected to directly and easily derive the time, energy, and memory footprints required to run their programs. This state of affair makes research into algorithmic complexity and automated cost analysis a pragmatic topic of investigation.

A good point to begin a focused overview of the relevant corpus is the work of R. E. Tarjan on *amortized computational complexity*. Of interest to us is the term "amortized" of this nomenclature. Taken directly from the world of finance, to *amortize* a cost is to substitute a bulk payment into a sum of smaller payments done ahead of time which, when accumulated, cover the cost of the actual payment without inducing the corresponding large debit. For example, if one expects to have to pay 1200€ at the end of the year, one can put aside 100€ a month and pay the whole sum without having to handle the larger financial stress associated with a 1200€ debit line. Let us dive in:

2.1 Amortized algorithmic analysis

The canonical reference here is[75]. In this paper, Tarjan introduces amortized complexity as a finer analysis tool than worst-case cost analysis, while still remaining safer than average-cost

complexity. Indeed, while the cost of a program is often close to the average, one exceptional case is enough to cause failure. Likewise, always assuming that each operation independently costs as much as it possibly can ignores all the possible optimizations performed thanks to the clever design of its programmers.

A good first example for Amortized analysis is the *functional queue*, implemented using two linked lists, `f` for the front of the queue, and `b` for its back. The front of the queue contains older elements, with the oldest one being on top, and the back contains the younger ones in reverse order, with the youngest on top. Adding a new item to the queue just pushes it on top of the back, and popping one just removes the topmost element from the front. If the front is empty though, the back of the queue is flipped, putting its oldest elements on top. This new list is then set as the new front of the queue. This is implemented as so in Ocaml:

```
type 'a queue = Q of ('a list) * ('a list)
let enqueue (Q (f,b)) x = Q (f,x::b)
let rec dequeue = fun
  | Q (x::f,b) -> (x, Q (f,b))
  | Q ([],[]) -> error (* no elements *)
  | Q ([],b) -> dequeue (reverse b, [])
```

Suppose we want to determine a bound for the time complexity for this queue data structure in terms of list operations (i.e. set a cost of 1 for each case analysis and constructor call on a list). The cost of reversing a list of length k is thereby $2k$.

With this metric, a naïve reasoning deduces that `enqueue` has a maximum cost of 1 per call, and `dequeue` a maximal cost of $2k + 4$, where k is the size of the back list. Therefore, if we process n elements, each being enqueued and dequeued, the size of the back is bounded by n, and we get a total cost $C(n)$ bounded by $1n + (2n + 4)n = 2n^2 + 5n = O(n^2)$, i.e. a quadratic time complexity bound in the worst case.

But an amortized analysis provides a lower bound in the same setting, in fact showing that this cost is bounded linearly. To do so, observe that each element is reversed exactly once: after the first reversing, it ends up in the front list, where it is never reversed further. This means that the calls to `reverse` over the entire lifetime of the structure are done on lists of

sizes $k_1, \ldots, k_p$ summing to exactly n. The complexity $C(n)$ is therefore bounded by:

$$C(n) \leq 1 \times n + \max_{k_1+\cdots+k_p=n} \sum_{1\leq i\leq p} (2k_i + 4) = n + \sum_{1\leq i\leq n} (2 \times 1 + 4) = 7n = O(n)$$

Therefore, the time complexity of processing n elements is always linear. This can more simply be accounted for by imagining that each element added to the queue is endowed with 7 credits. One of those credit is spent for the enqueuing. Reversing the back queue with $k > 0$ elements has cost $2k + 4$, but each element in the back possesses 6 time credit still, which means they can afford to be reversed "out of pocket" without occurring a time spending. In the worst case scenario (when the back contains a single element), the credits cover exactly the required cost. This means that crediting the n elements with a total of $7n$ credits allows the entire processing to occur without further costs, and that the worst-case time complexity is bounded by $7n$. Accounting for the behavior of the data structure over its entire lifetime, and loading its elements with credits accordingly, we were able to obtain a significantly better bound than was achievable via a naïve analysis.

Note that obtaining this tighter bound requires understanding an invariant about the stack: its size is split between front and back, and each element is processed identically. To understand how to derive those invariants and the tighter bounds they induce, we first present some notations, which we'll use throughout the manuscript.

2.2 Resource semantics for amortized analysis

Let $c \in \mathcal{C}$ be programs and let a deterministic, binary relation $c \to c'$ on programs denote a *small-step* reduction (i.e. program c becomes c' during evaluation *without* any intermediate states). A full execution of a program is then a (potentially infinite) sequence $(c_i)_i$ with $c_i \triangleright c_{i+1}$.

We lastly assume the existence of a function $k : \{(c, c') \in \mathcal{C}^2 \mid c \to c'\} \to \mathbb{Z}$, which associates a cost $k(c, c')$ to a program transition $c \triangleright c'$. Note that k can be negative, in which case it denotes a reclamation of resources. This can occur most importantly if the resource in question is memory, but not if it is time or energy (for those later cases, we can set $k \in \mathbb{N}$). The triplet $(\mathcal{C}, \triangleright, k)$ is a *resource semantic for* $\mathcal{C}$.

Given a reduction sequence $(c_i)_i$, the sequence $(k(c_j, c_{j+i}))_j$ is called its *resource profile*. We define K_i, the *footprint at step i*, as the partial sum $\sum_{0 \leq i < n} k_i$. Finally, the overall *footprint of the execution* of a program is the maximum of the footprints over all step $\max_i K_i$ (or $+\infty$ if no finite maximum exists). Note that the overall costs isn't always realized as the footprint at the last step: if one of the k_i is negative, then K may be realized at an intermediate step (think of a program that allocate a large amount of memory for its internal operations, then frees it before it finishes).

With this formalism, the cost $k(c, c')$ of a reduction step is determined dynamically. Using small-step semantics, it is sometimes possible to determine this one-step cost at compile-time, which unlocks a useful simplification which we now introduce. It consists in extending the original set of programs $\mathcal{C}$ with a new construct called a *tick*. Such tick, written tick(c,k), reduces to c in one step with cost k.

Formally, given programs with a resource semantics $(\mathcal{C}, \triangleright, k)$, a one $(\mathcal{C}', \triangleright, k')$ is defined by setting $\mathcal{C}'$ the smallest set (least upper bound, or l.u.b.) containing $\mathcal{C}$ and all $\text{tick}(c', k)$ for $c' \in \mathcal{C}'$, endowed with the accordingly modified reduction and cost metric:

$$\begin{aligned} \mathcal{C}' &= \textit{l.u.b. of map } \mathcal{C}' \mapsto \mathcal{C} \cup \mathcal{C}' \cup \{\text{tick}(c', k) \mid c' \in \mathcal{C}', k \in \mathbb{Z}\} \\ (\triangleright) &= (\triangleright) \cup \{(\text{tick}(c', k), c') \mid c' \in \mathcal{C}'\} \\ k'(c'_1, c'_2) &= \begin{cases} k & \textit{when } c'_1 = \text{tick}(c'_2, k) \\ 0 & \textit{otherwise} \end{cases} \end{aligned}$$

Then, we say $(\mathcal{C}, \triangleright, k)$ has *statically determinable step costs* if there is a map $\varphi : \mathcal{C} \to \mathcal{C}'$ such that all $c \in \mathcal{C}$ and $\varphi(c) \in \mathcal{C}'$ have the same resource profile, that is, its costs metric can be faithfully internalized using ticks. In this manuscript, we shall only consider programs with statically determinable step costs.

2.3 The banker's and physicist's method

The goal of amortized analysis is to find, given a resource semantics and a family of programs with varying inputs, a common upper bound for their footprints. For example, the family of programs for the previously introduced stack structure contains all sequences of n stack operations, and we derived an upper bound $7n$ for their footprints. Tarjan provides two systematic ways to derive such bounds over families of programs. Those two (equivalent)

methods, the "banker" and "physicist" formalisms, are strictly equivalent. First, let us begin by describing the *banker's method*.

The banker's method formalizes the idea of "saving now to pay later" . Formally, a new *amortized* cost semantics k' is devised which satisfies $K'_i \geq K_i$. This suffices to cover the footprint at each step, and therefore guarantees $K' \geq K$. If this new cost semantics is sufficiently simple, bounds can be computed closed forms. For example, the amortized semantic for stacks we informally described assigns a cost of seven units per `enqueue` operation, and zero for `dequeue` operation. Using the stack invariant, it is proved that this semantics indeed gives an upper bound of the actual time footprint. Here, the "simplicity" requirement is satisfied: a program with n operation has footprint at most $7n$, the upper bound being realized with n `enqueue` and no `dequeue`. The banker's method is well-adapted to situations in which relevant invariants are already well-understood.

When automated methods are required, it may not be desirable to derive a whole new cost semantics for the specific family of program under consideration. Thankfully, an equivalent characterization of amortization preserves the old semantics: the *physicist's* method. In the physicist's method, each program $c \in \mathcal{C}$ is given a *potential* $\Phi(c) \in \mathbb{N}$ such that $\Phi_i - \Phi_{i+1} \geq k_i$. This last equation is the *potential condition*, and encodes the fact that the loss of potential during evaluation covers for the program's footprint. We write $\Phi_i = \Phi(c_i)$ for the potential of an intermediate step when unambiguous. The following calculation shows that initial potential Φ_0 of a program is an upper bound for all its intermediate footprints, and therefore its total footprint:

$$K_i = \sum_{0 \leq j < i} k_j \leq \sum_{0 \leq j < i} (\Phi_i - \Phi_{i+1}) = \Phi_0 - \Phi_n \leq \Phi_0$$

For the queue example, setting $\Phi_0 = 7n$ suffices to guarantee $\Phi_i - \Phi_{i+1} \geq k_i$ by the queue invariant. In general, one can first pose $\Phi_0 = f(n, \vec{\alpha})$ for some free parameters $\vec{\alpha}$, find a subset of those parameters which satisfy the potential condition, and finally minimize Φ_0 by varying the $\vec{\alpha}$. This method provides the skeleton of automated amortized computational complexity analyses.

To finish up this introductory exposition, we show that the two methods are strictly equivalent: one defines a banker's cost semantics K' from a potential Φ by setting $K'_i = \Phi_0 - \Phi_i$ and conversely, every banker's cost semantics K' defines a potential by $\Phi_i = K'_i - K_i$. This justifies

that, in full generality, to find an upper bound for the amortized cost of a family of programs reduces to finding a potential.

2.4 Early days of Automated Amortized Resource Analysis

Over the last twenty years (from 2003 onward), the theoretical framework of *Automated Amortized Resource Analysis* (hereafter AARA) was developed as a family of increasingly syntax-directed and precise type systems for functional programs that implements automated complexity analysis with the potential method[34].

The seminal paper here is Martin Hoffmann's 2003 paper[37] which we briefly mentioned in the introduction. Its results are eloquently summarized in a short abstract, which we reproduce here in its totality:

> *"We propose a linear type system with recursion operators for inductive datatypes which ensures that all definable functions are polynomial time computable. The system improves upon previous such systems in that recursive definitions can be arbitrarily nested; in particular, no predicativity or modality restrictions are made"*

The λ-calculus considered here has affine static semantics with functions, both pattern-matching and projection pairs, and a predefined set of type operators (notably, homogeneous lists, labeled binary trees, and binary-encoded integers) that come with their own structural recursion primitives. The notion of "time" involved is not based on a particular model. Instead, it arises as some integer-valued metric on reduction in another unspecified formalism. This metric is constrained as to encode a sensible cost to the functional programs when they are compiled to combinator algebra.

The main ingredient guaranteeing that execution time is indeed polynomial is a token type $\Diamond$ that is necessary to introduce and eliminate terms of each type. This encodes spending some currency for program operations. For example, pushing an element of type A to a list oftype $L(A)$ is an operation of type $\texttt{cons} : \Diamond \multimap A \multimap L(A) \multimap L(A)$, which means it has unit cost.

Correspondingly, structural recursion destroys constructors and frees a token at each step, which can be used to finance further computations. As a result of this requirement, programs in Hoffmann's formalism which can be typed without $\Diamond$ are *non-increasing in size*, which in

turns means they either diverge by looping forever or converge in polynomial time. Since the λ-calculus under consideration is terminating, polynomial time is thereby ensured.

This result relies on two language features which makes it unsuitable as-is for resource analysis of functional languages, which was overcome in further work:

1. Program source code is expected to explicitly manipulate tokens. This departs from the usual programming style in ML languages, in which resources such has allocations for constructors need not be dealt with.
2. Data structures (lists and trees) have to be specified manually, together with their token-aware construction and recursion primitives. The correctness of the method relies on the specific definition used and is not generalized to general-purpose algebraic data types.

Nevertheless, this promising first result laid the groundwork for the automated resource analysis developed by Martin Hoffmann before his unfortunate passing, together with his student Jan Hoffmann, who continues the work up to this day.

2.5 The AARA formalism for static analysis

Jan Hoffmann's work extended the result of Martin Hoffmann to an inference procedure for ML-style languages, allowing polynomial bounds to be determined, first for a class of program similar to those we mentioned in the previous section, and then for more mainstream programming constructs. We describe in this section the formal setup this analysis uses, in order to focus on more technical points in the next section. A bird's eye view of AARA is available in[34], and detailed proofs in J. Hoffmann's PhD thesis[33].

AARA takes the form of a type system, whose typing relation is extended with static bounds for the potential before and after evaluation of the typed expression. Those judgments are then shown to be sound regarding a resource-aware semantic for the language. Bounds are obtained by refining a preliminary typing derivation of simply-typed λ-calculus programs with resource-aware types.

Those resource-aware types are created by associating, to each node of a given type expression, a potential annotation partially specifying the changes in potential incurred by introducing

and eliminating that node. Those annotations are gathered at the root of the type expression. We choose to keep those them unspecified until the next section: all annotations in this section are replaced with the placeholder $\bullet$, and all resource-aware types are written $A^\bullet$, where A is a simple type.

At compile-time, an expression e of type $A^\bullet$ holds some potential $\Phi(e : A^\bullet) = \Phi_A(\bullet)$. The function Φ_A takes in annotations and returns a polynomial expression for the potential of the expression, whose free variables encode the yet-unknown sizes, length, depth, etc. of the runtime value of e. Symbolic reasoning on those expressions allows resource information to flow through the analysis.

In technical terms, AARA typing judgements and reduction relations respectively take the form

$$\Gamma^\bullet \overset{q}{\underset{q'}{\vdash}} e : A^\bullet, \text{ and } V \overset{m}{\underset{m'}{\vdash}} e \rightsquigarrow v, \text{ where}$$

- e is an expression of resource-aware type $A^\bullet$, which reduces to v;
- V is a runtime environment, and $\Gamma^\bullet$ a context of resource-aware types approximating it;
- and, m and m' are runtime costs encoded in the banker's way, and p and p' are static potential annotations providing sound bounds for them.

Then, the associated soundness theorem states that, if the typing judgment above holds, and furthermore V is any runtime environment typeable with $\Gamma^\bullet$ (i.e. each value in it typechecks with the associated resource-aware type) then the resource-aware reduction relation above holds as well for any values of m and m' such that:

- $m \geq q + r + \sum_{(v':A'^\bullet)\in V} \Phi_{A'}(\bullet)$,
- $m' \geq q' + r + \Phi_A(\bullet)$,
- and r is some amount of resources untouched by the evaluation of e.

The typing rules are defined on programs in administrative normal forms: all sub-expression are named. Furthermore, programs are assumed to be syntactically linear: a named value is used exactly once, and specific syntactic construct allow all variables to be explicitly shared and given a vacuous use to satisfy linearity. An automatic pre-processing phase transforms

programs into one satisfying those two conditions. Let us give some example of typing rules to finish up this presentation of the core of AARA. We shall focus on pairs (figure 2.1) and functions (figure 2.2). The rule for pair formation (AARA-PAIR) merely states that when two components x, y of a pair are already evaluated, the additional cost of evaluating the pair (x, y) is equal to some constant q_{pair}. When pattern-matching on a pair, the potential of each component is released from the pair, with an offset of q_{unpair}. Note that there is in general no reason that the potential associated to each component to be equal before packing and after unpacking, and that the overall potential of the expression may have changed in between the use of the two rules.

$$\frac{q + \Phi_A(\bullet) + \Phi_B(\bullet) \geq q' + \Phi_{A\times B}(\bullet) + q_{\text{pair}}}{x : A^\bullet, y : B^\bullet \overset{q}{\underset{q'}{\vdash}} (x, y) : (A \times B)^\bullet}\ (\text{AARA-PAIR})$$

$$\frac{\Gamma^\bullet, x : A^\bullet, y : B^\bullet \overset{p}{\underset{p'}{\vdash}} e : C^\bullet \qquad \begin{cases} p' \geq q' \\ q + \Phi_{A\times B}(\bullet) \geq p + \Phi_A(\bullet) + \Phi_B(\bullet) + q_{\text{unpair}} \end{cases}}{\Gamma^\bullet, z : (A \times B)^\bullet \overset{q}{\underset{q'}{\vdash}} \textbf{let } (x', y') = z \textbf{ in } e : C^\bullet}\ (\text{AARA-UNPAIR})$$

Figure 2.1: *AARA* typing rules for pairs

When extending those rules to iso-recursive, tree-shaped, algebraic data-types – that is, types defined by a recursive equation $T = A_1 \times T^{k_1} + \cdots + A_n \times T^{k_n}$, each constructor is given a separate direct cost, and pattern matching branches are analysed separately. Then, each branch's cost is over-estimated by a free cost variable as to line up with the other ones, which may cost more.

Functions (potentially recursive) are the trickier case of the analysis, and require some restrictions. First, currying is not made resource aware: the type system assumes potential is only manipulated on saturated functions calls.[1] Secondly, closures (which are fully synonymous with functions in this context) may not hold potential: indeed, the potential of the values they capture may change during evaluation. Furthermore, the closure's potential is itself

[1]This limitation and its justification turns out to manifest themselves in a clearer way in the system we introduce in this manuscript. For the moment, we hope the patient reader can take the well-foundedness of this requirement at face value.

liable to change due to external and recursive calls. As such, it is not possible to derive a safe, non-empty potential for unrestricted closures.

Function terms have syntax **fun** $f(x) = e$, which extends the ML-style syntax **fun** $x \to e$ to recursive definitions. Resource-aware function types are not annotated in the same way as data structures. Not only do they hold the usual $\bullet$ annotation when they appear at the top-level of a type expression, they also include an annotation context $\Theta^\bullet$ giving annotations for their arguments and result, and bear potential annotations q and q', denoting potential before and after its calls. As such, a fully-annotated function type is $(A \underset{q'}{\overset{q}{\to}} B, \Theta^\bullet)^\bullet$. This, together with the previously stated limitations, and the type-level annotation on argument and return types, immediately give the rules for function calls and definitions, so we only show the rule for abstractions. Here, the annotation context $\Theta^\bullet$ is built immediately by asserting that the annotated types $(A^\bullet, B^\bullet)$ encoding input-output pairs of resource annotations exists. Note that the two potential annotations in the function's type are unified with those of the function's body, definition, avoiding cyclicity in the typing rule.

$$\frac{\Gamma^\bullet, \left(f : A \underset{p'}{\overset{p}{\to}} B, \Theta^\bullet\right), x : A^\bullet \underset{p'}{\overset{p}{\vdash}} e : B^\bullet \qquad \begin{cases} \Phi_\Gamma(\bullet) = 0 \\ (A^\bullet, B^\bullet) \in \Theta^\bullet \end{cases} \qquad q \geq q' + q_{\text{abs}}}{\Gamma^\bullet \underset{q'}{\overset{q}{\vdash}} \textbf{fun}\ f(x) = e : \left(A \underset{p'}{\overset{p}{\to}} B, \Theta^\bullet\right)^\bullet} \text{(AARA-ABS)}$$

Figure 2.2: AARA: typing rules for abstractions

The case of high-order functions is a little more involved, as the resource behavior of a functional argument influences the behavior of the high-order function using it. Such functional arguments require a constraint on potential annotation to be defined at a function's site of definition, carried in their types, and finally instantiated in the type of the high-order functions that uses them. This complexity only compounds as the order of functions increases, or as recursion is brought into play. We will give more thought about those cases in the next section, where we explore the mechanisms implementing the all-important potential annotations $\bullet$ and their semantics function $\Phi_A(\bullet)$ in real-life analyzers.

2.6 Type-level resource annotations for AARA

This section's goal is to answer the questions left by the previous one regarding resource annotations on types and functions: how are the $\bullet$ defined? How are they instantiated in the type system? What is the algorithm behind the Φ potential function? What kind of algorithms can be tightly bound with it? To answer those, let us begin by presenting a general principle, *uniform recursion*, and give an account of the annotations and potentials for linked lists. This section has to be more technical than the preceding ones, as is presents the core of the algorithmic and combinatorial reasoning of AARA type systems.

Uniform iteration When an algorithm loops or recurses, it often does so uniformly for all patterns in a given data structure. This is for example the intended meaning behind Python's `for x in y` loop, OCaml's `List.iter` and Haskell's `Foldable` typeclass. In general, a pattern is not merely a node in a data-structure, but may be a predetermined set of node directly or indirectly related in a data structure. For example, pairs (l_i, l_j) of elements in a list l in order of appearance (indices $i < j$), or all inner nodes of a tree with a label a under a node labeled b. Potential in AARA is built up from enumerations of such patterns in *polynomial datatypes*, that is, simple types T defined by constructors, each of those taking arguments which may be themselves of type T. Formally, those are exactly the smallest solutions to equations of the form:

$$T \simeq A_1 \times T^{k_1} + A_2 \times T^{k_2} + \cdots + A_n \times T^{k_n}.$$

Example: sorting linked lists Let us see how those annotations enumerating uniform iterations can be used to derive potential for list algorithm. Recall that linked lists are the polynomial type $L(T) = 1 + T \times L(T)$. The patterns of nodes within a list $[l_1, \ldots, l_n]$ are therefore tuples of lists elements $(l_{i_1}, \ldots, l_{j_m})$ with of increasing index: $0 \leq i_1 < \cdots < i_m \leq n - 1$. The number of such patterns of size m in a list of length n is immediately expressible with as a binomial $\binom{n}{m}$.

As an example, consider the *select-sort* algorithm on linked list in OCaml shown in figure 2.3. This algorithm will iterate on all pairs of items on the list, and we have added *print* statements to represent the costs of nested iterations. Namely, the cost of operation per pair of elements is q_2, the cost for processing one element is q_1, and the unitary cost for the entire

```
let select_sort l =
  let rec select large min rest = (* extract the minimum element *)
    match rest with
    | [] -> (min, large)
    | y::ys ->
    print_endline "This cost q2.";
    if min < y then
        select (y::large) min ys
      else
        select (x::large) min ys in
  let rec sort l = (* put the minimum at the head of the list *)
    match l with
    | [] -> []
    | x::xs ->
    print_endline "This cost q1.";
      let min, large = select [] x xs in
      min :: (sort large) in
  print_endline "This costs q0."; (* <= entry point for select_sort *)
  sort l
```

Figure 2.3: *OCaml* listing of the `select_sort` algorithm

function is q_0. In this setup, the total cost $C(n)$ of sorting a list of length n is

$$\begin{aligned} C(n) &= q_2\binom{n-1}{2} && + q_1\binom{n}{1} + q_0\binom{n}{0} \\ &= q_2\frac{(n-1)(n-2)}{2} && + q_1 n + q_0 \\ &= \frac{1}{2}q_2 n^2 && + (q_1 - \frac{3}{2}q_2)n + q_0. \end{aligned}$$

This computation shows that bounding the cost of nested iteration can be separated into (1) bounding the cost of each immediate step, (2) finding out on which pattern of which data each particular step is repeated on, and (3) summing those costs using combinatorial identities. The AARA annotations allow step (1) and (2) to be performed in the type system for all polynomial datatypes in such a way that step (3) simplifies to a combinatorics problem.

Identifying patterns Let us assume we have a datatype T defined by $T \simeq A_1 \times T^{k_1} + \cdots + A_n \times T^{k_n}$. This is written in AARA notation as $T = \langle C_1 : (A_1, k_1), \ldots, C_n : (A_n, k_n)\rangle$.

AARA inductively defines for each such datatypes a family of *base indices* which encode which pattern in a value of that type is being iterated over at a point in a nested iteration.

To do so, for each pair of values $v, u : T$ values, we write $v \leq u$ if u is a subtree of v[2]. In this situation. The base indices $I_T$are defined inductively:

- There is a trivial base index $I_T = \star$ encoding absence of iteration
- If $T = T_1 \times \cdots \times T_k$ is a tuple type and $I_{T_1}, \ldots, I_{T_k}$ are base indices for each component, then $I_T = (I_{T_1}, \ldots, I_{T_k})$ is a base index.
- For each tuple of constructors $(C_{i_1}, \ldots, C_{i_m})$ and base indices $I_{A_{i_1}}, \ldots, I_{A_{i_m}}$ for the arguments of each of them, there is a base index $I = \left\langle (C_{i_1}, I_{A_{i_1}}), \ldots, (C_{i_m}, I_{A_{i_m}}) \right\rangle$.
- The base indices $\star$, $(\star, \ldots, \star)$, and $\langle\rangle$ are identified

The patterns it iterates over are also defined inductively: Given a value $v : T$ and a base index I_T for its type, it matches the following sub-structures of v:

- If $I_T = \star$, it matches on v itself.
- If $I_T = (I_{T_1}, \ldots, I_{T_k})$, then $v = (v_1, \ldots, v_k)$, and I_T matches on all tuples $(u_1, \ldots, u_k)$ where each u_i is a sub-structure of v_i matched by I_{T_i}.
- If $I_T = \left\langle (C_{i_1}, I_{A_{i_1}}), \ldots, (C_{i_n}, I_{A_{i_n}}) \right\rangle$, it matches over all tuples $(b_1, \ldots, b_m)$ such that there is a sequence $t \geq t_1 > \cdots > t_m$ of subtrees of v ordered by inclusion, such that $t_j = C_{i_j}(a_j, \ldots)$ for some $a_j : A_{i_j}$, and each b_{i_j} is a sub-structure of a_{i_j} matched by $I_{A_{i_j}}$.

Potential annotations To each base index, AARA associates a potential coefficient in $\mathbb{Q}$. This means that for all non-function types T the annotated type $T^\bullet$ is just a pair (T, Q_T), where $Q_T = (q_{I_T} \in \mathbb{Q})_{I_T \in \mathcal{I}_T}$ is a finite family of rationals indexed by base indices in $\mathcal{I}_T$. Remember that types aren't *deeply* annotated: annotations only appear at the top-level of types, never in the inner nodes of type expressions.

This leaves the case of annotated function types. If $(A \xrightarrow[q']{q} B, \Theta^\bullet)$ is an annotated function type, then $\Theta^\bullet$ is merely a finite set of tuples $(Q_{|A|}, Q_{|B|})$ of annotations to be instantiated on

[2] *Subtrees* here are contiguous subtrees

the argument and return expressions of a call site of the function. The operation $|T|$ on types merely erases all type annotations of the arguments and return type. This guarantees that intractable higher-order functional arguments do not influence the potential estimate. Finally, the potential of a value according to its annotated type $\Phi(T, Q_T)(v)$ is defined as a sum over the family $Q_T = (q_{I_T})_{I_T \in \mathcal{I}}$ by setting:

$$\Phi_T(Q_T)(v) = \sum_{I_T \in \mathcal{I}} q_{I_T} \times \text{card } \{u \mid I_T \text{ matches on } u \text{ in } v\}$$

This concludes the presentation of AARA analysis. The potential annotations it implements are tailored to polynomial complexity. Indeed, given the definition of base indices and potential, all expression involved in annotation are polynomials, whose free variables are the number of sub-structures within its free variables. While different formalizations of potential have been developed, only the polynomial ones described above are fully implemented. Furthermore, the RAML[3] tool developed by Hoffmann refines this type system with a separation between stack and heap resources of resources to better model memory allocations for OCaml.

2.7 Dal Lago anda Gaboardi's dℓPCF

As we've seen in the previous section, AARA infers resource footprints of programs through the enrichment of simple types with indices that approximate resource manipulation at runtime. This requires analyses to implement *reasoning over index expressions*. The bespoke system created for polynomial AARA has been successfully implemented, but its diverse extensions to high-order functions, shared closures, logarithmic/exponential complexities, etc. are not so straightforwardly implementable. This is a first, immediate obstacle to the design and implementation of extensible and reusable resource analyses.

More recent work involving U. Dal Lago, B. Petit, and M. Gaboardi[20, 21] regarding complexity analysis within type system allows those indices to be expressed in the formalism of *constrained type systems*. Those systems are parameterized by a "constraint judgement", that defines which values can be assigned indices in types and programs by means of a logical constraint (often in a fragment of first-order logic). Those type systems parameterized by a first-order constraint enable mainstream techniques from constraint solving and tools such as SMT solver ("Satisfaction Modulo (a first-order) Theory") to be used for those analyses.

[3] raml.co

Indices A significant model in this family is Dal Lago & Petit's dℓPCF[21], which is a parameterized type system that can – in theory – infer the number steps required to evaluate a PCF term in the call-by-value SECD machine[51] or KAM call-by-name machine[49]. Recall that PCF is the simply-typed λ calculus with integers and fixpoints, with the following grammar:

$$t, u, v ::= x \mid n \in \mathbb{N} \mid t + 1 \mid t - 1 \mid \lambda x.t \mid t\, u \mid \texttt{fix}x.t \mid \texttt{ifz}\ t\ \texttt{then}\ u\ \texttt{else}\ v$$

Note that since PCF is a Turing-complete language, and as such bounding the number of evaluation steps, or even deciding when a program finishes, is a Turing-complete problem. The authors nevertheless achieve an inference procedure *up to* a decision procedure for a first-order theory. It is understood that such an oracle may not exist for all dℓPCFprograms, while classes of programs with sufficiently "tame" recursions are analyzable.

The type-level index language introduced in dℓPCFincludes variables a for natural integers, a fixed collection of n-ary functions f to and from the integers (including primitives for constants, addition and bounded subtraction), a bounded sum over integers Σ, and a special *forest cardinality operator* $\triangle$ whose meaning will be clarified later on:

$$I, J, K ::= a \mid f(\vec{I}) \mid \sum_{a<I} J \mid \triangle_a^{I,J} K$$

Constraints Indices are manipulated through a finite conjunction Φ of inequalities constraints $I \leq J$. The free variables in Φ are scoped from in a *index scope* φ. Note that, in the natural integers, the relation $(\leq)$ also defines the relations $(=), (\neq), (<), (>)$, and $(\geq)$. The semantics of those constraints are given by a *valuation* ρ that provides a value $\rho(a) \in \mathbb{N}$ to each index variable in φ, and a *equational program* $\mathcal{E}$ that provide a partial function for each function symbol f in the index language. The semantics of an index expression are given (when defined) by the integer $[\![I]\!]_\rho^{\mathcal{E}}$ which is defined in the natural way. Putting those ingredients together, dℓPCFdefines a semantic judgement on indices which can appear in the type system:

$$\varphi; \Phi \vDash_{\mathcal{E}} I \leq J \text{ iff for every } \rho \text{ satisfying } \Phi,\ [\![I]\!]_\varphi^{\mathcal{E}} \leq [\![J]\!]_\varphi^{\mathcal{E}}$$

.

Types This semantic judgment on indices enables the dℓPCFtype system to delegate solving index constraints onto a dedicated oracle distinct of the type system. As such, there is an inference procedure that provides an indexed type to programs, but may not be able to instantiate some indices should the oracle fail to solve the constraint. In any case, bounding the number of evaluation steps boils down to bounding the number of shared copies of each recursive value $\texttt{fix}\, x.t$ that may appear in the code, whose calls may themselves cause more copy of themselves to be created, etc. This is done with *linear types* and *modal* shareable types. For call-by-value semantics, integer values are types $\mathrm{Nat}[I, J]$ for an integer between I and J, and are freely shareable. In call-by-name mode, integers are also linear, as a lazy integer may carry some unevaluated terms would change the program's cost if shared. Functions are linear (not shareable) but can take in shareable arguments and return shareable results. Lastly, shareable closures track the maximum number of copies a closure may have over the lifetime of the program. Tracking this information allows the type of arguments and results of functions to be parameterized according to the execution of the program. For example, the bounds of integers in function results can depend on the number of iterations required to produce them. The syntax for types is therefore:

$$A, B ::= \sigma \multimap \tau \qquad \text{(linear types)}$$
$$\sigma, \tau ::= [a < I].A \mid \mathrm{Nat}[I, J] \qquad \text{(shareable types)}$$

Sharing The shareable type $[a < I].A$ is a closure value of type A that can be shared up to I times. The index variable a tracks the number of copies, and is bound in A. This means shared functions "know" at compile-time which copy they are. For example, let us consider some function f to/from integers which, that, at runtime, makes five recursive calls to itself. Let us assume that the first recursive call requires one copy of f, that the second one calls f twice, that the third one calls f thrice etc., and that no further calls occur. In total, a call to f will therefore need $1 + 2 + 3 + 4 + 5 = 15$ intermediate calls over its five recursions, and sixteen when counting the root call. Then, f can be typed as:

$$f : [a < 16].\mathrm{Nat}[I, J] \multimap \mathrm{Nat}[I', J']$$

Abbreviating $A = \mathrm{Nat}[I, J] \multimap \mathrm{Nat}[I', J']$, This type can be thought of as a collection of copies of f, each of them indexed by the number of times they are used in intermediate calls, and

each of them recording their place in the evaluation: the root call has index $a_0 = 0$, the first copy has index $a_1 = 0$; the second $a_2 = 1, 2$; the third one $a_3 = 3, 4, 5$; etc.

$$f : A \otimes [a_1 < 1].A \otimes [a_2 < 2].A[a_2 + 1/a] \otimes \cdots \otimes [a_5 < 5].A[a_5 + 10/a]$$

Finally, when all sharing is expanded, f can be typed equivalently with

$$f : A[0/a] \otimes A[1/a] \otimes \cdots \otimes A[15/a]$$

The sharing induced by inner recursive calls is the most subtle data that must be tracked to obtain cost bounds in dℓPCF, and the one source of potential undecidability. This tracking is done by the *forest counting* operator $\bigtriangleup_a^{I,J} K$ on indices. To understand its behavior, consider a recursive function $f = \texttt{fix}\, f'.t : B$, where f' is bound in the function body t. This function may be shared, and each shared copy used to start one "root" call to f. Then, each call to f may in turn start more intermediate call, who themselves call f, etc. Each call to f is therefore the root of a *tree* of intermediate calls which describes the iterated expansions of the body t of f. All those root then together produce a *forest*. Assuming the program terminates, all those trees have finite depth and width, and the forest has finitely many trees. Then, to know the full cost induced by all calls to f, it suffices to know the immediate costs of each call, and the number of node in the forest. The $\bigtriangleup_a^{I,J} K$ index counts those nodes in by identifying each node with an integer, starting at 0, and counting in the top-most left-most order of nodes. Namely:

- If J is the number of trees in the forest;
- and I is the number of nodes of the forest already visited;
- and K, with free variable a, is the number of children of the a^{th} node of the forest;
- then $\bigtriangleup_a^{I,J} K$ denotes the total number of node to visit in the whole forest.

Note that this number may fail to be defined for certain parameters I, J, K, and it is Turing-complete to determine if the forest counting index associated to f is indeed finite. Since implementing the forest-counting index is out of the scope of this thesis, we shall stop its description here, and redirect the reader to[21] for more information.

Type inference The type inference procedure for dℓPCFis based around the constraint judgement $\varphi, \Phi \models_{\mathcal{E}} I \leq J$, in which we ask that some model $\mathcal{E}$ of index operations guarantees that $I \leq J$ is satisfied given some hypotheses Φ. This judgment is the leaf node of typing derivations. The main typing judgment and its subtyping judgment, both shown below, ensure that the term t of type T in scope Γ evaluates to a normal form in at most K steps, with φ, Φ and $\mathcal{E}$ keeping their previous meanings. Subtyping is standard, and uses reasoning on indices to relax bounds in integers and amount of sharing in closures.

$$\varphi, \Phi, \Gamma \vdash^{\mathcal{E}}_{K} t : T \quad \varphi, \Phi \vdash T \sqsubseteq T'$$

This type inference procedure remains simple, as the Turing-complete reasoning on programs it may encode is delegated to the oracle E. Recall that $\mathcal{E}$ resolves each index primitive into a partial function $\mathbb{N}^k \rightharpoonup \mathbb{N}$. So far, its only significant use was in counting forest cardinalities. This is extended in the following manner: when instantiating an index I in some scope $\varphi = \vec{J}$ in a typing derivation, we assume $\mathcal{E}$ contains some predefined function $f_I : \mathbb{N}^{|\varphi|} \rightharpoonup \mathbb{N}$ representing I and unify $I = f_I(\vec{J})$ in the entire typing derivation. This creates a global collection of index functions $f_I, f_J, \ldots$, and instantiates all constraints judgements as:

$$\begin{aligned}
&\vec{K}, \Phi \vdash_{\mathcal{E}} I \leq J \\
&\rightsquigarrow \vec{K}, \Phi[f_I(\vec{K})/I, f_J(\vec{K})/J] \vdash_{\mathcal{E}, f_I, f_J} f_I(\vec{K}) \leq f_J(\vec{K}) \\
&\rightsquigarrow \forall \vec{K}. \Phi[f_I(\vec{K})/I, f_J(\vec{K})/J] \implies f_I(\vec{K}) \leq f_J(\vec{K})
\end{aligned}$$

Performing that transformation reduces the problem of typing the PCF terms for their time footprint to a problem of instantiating the function symbols f_I under a finite number of first order constraints of the form $\forall \vec{K}. \Phi \implies f_I(\vec{K}) \leq f_J(\vec{K})$, where Φ is itself a finite conjunction of inequalities $f_{I'}(\vec{K}) \leq f_{J'}(\vec{K})$. Note that the forest cardinality operator may occur in the constraints, and as such, if the scope of the problem is not reduced, no hope of automatic solving can be had. Nevertheless, this approach is in our point of view quite welcome, as it realizes the same kind of enriched type inference procedure as AARA while using standard first-order constraints and relates resource analysis to mainline topics of research in type inference (linear logic, dependency).

2.8 AARA with constraints: λ_{amor}

Given the similar concerns of both dℓPCFand AARA, as well as the generality of the index system the former introduced, it is far from surprising that an encompassing system extending dℓPCFhas emerged, implementing the potential method in this new formal setting. This is exactly what V. Rajani, M. Gaboardi, D. Garg and J. Hoffamm achieved in a 2021 POPL paper[69], where they introduce λ_{amor}. Technically, they introduce a family of formalisms, parameterized by a higher-order signature for indices, that provide a type system capable of reasoning over the notion of resources that this signature encodes. The main features of λ_{amor}coming together to implement amortized analysis are (1) indexed types *à la* dℓPCFallowing sizes and structures to be accounted for at type-level; (2) a potential-bearing type constructor allowing arbitrary types to hold potential parameterized by their own indices; (3) a cost-bearing construct that allow monadic computations to be guarded by spending potential; and (4) linearity, which guarantees potential and costs aren't spuriously duplicated or omitted. With this setup, λ_{amor}focuses on the theoretical side of the resource analysis problem. As such, the authors don't provide an implementation, and do not take into account recoupable cost, opting for a simpler formalism in which costs are merely accumulative.

Indices The indices consist of natural and positive real numbers with additions and subtractions, abstraction and application of index-level functions, and iterated sum $\sum_J \lambda(a : \mathbb{N}).I$ with semantics $\sum_{0 \leq a < J} I$. Constraints on indices are conjunctions of equalities and strict inequalities (both restricted to numbers). Finally, types in $\lambda_{\text{amor}}$are either base types or types depending on any number of indices (both number and functions allowed). The type system is polymorphic. For example, consider lists of length n whose elements are of type type A and havesizes $l(0), l(1), \ldots, l(n-1)$ respectively. The type of the i^{th} element will then be $A(l(i))$. A `map` function over such lists needs to be aware of those sizes to be well-typed. This requires its functional argument passed as a polymorphic function over the index i:

$$\texttt{map} \ : \ \forall A, B : \mathbb{N} \to \text{Type}.\, (\forall i : \mathbb{N}.\, A(i) \to B(i)) \to \text{List}(n, A) \to \text{List}(n, B)$$

Types Reasoning on indices is implemented through three families of types: quantification over indices, abstraction and applications over indices, and witnessing and assuming the

validity of constraints. This gives the following fragment of the syntax of types:

$$T ::= \forall a.T \mid \exists a.T \mid \lambda a.\, T \mid T\, I \mid C \Rightarrow T \mid C\&T \mid \dots$$

Common data types are also implementable in λ_{amor}, another improvement over dℓPCF. Common linear types (lazy and eager pairs, linear sums and functions) are primitive, as well as shared references and lists indexed on lengths. The shareable closure of dℓPCFis also present under of the form of *sub-exponentials* $!_{a<I}T$ where the index variable a is bound in T. Further base types may be added as needed, such as integers. Note that the formalism developed in [69] only includes lists as data structures, and doesn't allow for algebraic datatypes.

$$T ::= \mathbb{1} \mid T \otimes T \mid T \oplus T \mid T \multimap T \mid T\&T \mid !T \mid !_{a<I}T \mid L^n(T) \mid \dots$$

This only leaves cost-bearing and potential-bearing types to be introduced. Those form a dual pair which encodes the evaluation order of programs in λ_{amor}.

Potential and costs Values of type T that bear a potential I have type $[I]T$. The index I is a $\mathbb{N}$-valued index expression made from the index variable in the ambient scope, which may also appear in T. Likewise, expressions whose evaluations spend I potential are typed $\mathbb{M}_I T$. A monadic structure is defined for $\mathbb{M}$, which is *graded* on costs, that is to say the parameter I vary in a suitable way in the types of the monadic *Return* and *Bind*. This is novel in λ_{amor}. This monadic structure allows larger computations to be built up from smaller ones while accumulating costs, and the ordering of bindings within monadic computations allows costs to be accumulated in a specific order. Dually, $[I]T$ is given a graded comonadic structure, encoding that $[I]T$ lives in an environment from which resources may be extracted in a given order (with primitives *Extract, Extend*). This is essential, as assigning then freeing potential is not equivalent to extracting then re-assigning it.

Potentials and costs interact to implement the potential formalism: a value of type T can be promoted to a computation with cost I which assigns this potential to I, typed as $\mathbb{M}_I[I]T$ (*Store*). Spending potential is encoded as compensating for costs: given a computation from A to B bearing cost $I + J$, and an input A bearing potential I, a resulting computation can be obtained which produces the B at cost J (*Spend*).

All those primitive can be obtained as simple macros over the monad/comonad primitives of the monad/co-monad pair introduced above. As to encode a program point that uses some

resources, a *Tick* primitive simply has cost I and returns a unit tuple of type $\mathbb{1}$. Finally, subtyping allows potential to be under-estimated and cost to be over-estimated (*Sub-[]*, *Sub-*$\mathbb{M}$), and linearity can be relaxed by erasing values with potential that can be wasted and computations that don't need to be run.

The related primitives giving potential and costs their (co)monadic structures are reproduced here. In our eyes, they finalize – with the index setup taken from dℓPCF– a translation of the automated amortized analysis into mainstream type systems for typed λ-calculi.

- **Cost monad**

$$\textit{Return}\ \ T \multimap \mathbb{M}_0 T$$
$$\textit{Bind}\ \ \mathbb{M}_I A \multimap (A \multimap \mathbb{M}_J B) \multimap \mathbb{M}_{I+J} B$$

- **Resource Comonad**

$$\textit{Extract}\ \ [0]T \multimap T$$
$$\textit{Extend}\ \ ([I]A \multimap B) \multimap [I+J]A \multimap [J]B$$

- **Amortization**

$$\textit{Tick}\ \ \mathbb{M}_K \mathbb{1}$$
$$\textit{Store}\ \ T \multimap \mathbb{M}_I [I]T$$
$$\textit{Spend}\ \ (A \multimap \mathbb{M}_{I+J} B) \multimap [J]A \multimap \mathbb{M}_I B$$

- **Relaxation**

$$\textit{Erase}\ \ A \multimap \mathbb{1}$$
$$\textit{Sub-[]}\ \ I \leq J \models [J]T \sqsubseteq [I]T$$
$$\textit{Sub-}\mathbb{M}\ \ I \geq J \models \mathbb{M}_J T \sqsubseteq \mathbb{M}_I T$$

Figure 2.4: Amortization primitives in λ_{amor}

2.9 AARA through Abstract Interpretation

Abstract interpretation[16] is a versatile and composable formalism for static analysis with a wide range of applications. As such, providing a thorough introduction to it falls well beyond the scope of this section. In a few words, abstract interpretation approximates the denotational semantics of values in a program in an *abstract domain*, a complete lattice whose order relation formalizes the notion of "being a more precise approximant". Abstract domains based on combinatorics approximate bitvectors and abstract domains based on systems of numerical equations approximate numerical values. Past this, tree-shaped structures can be endowed with abstract domains focusing on some aspect of the data structure, such as height. Abstract domains can furthermore be combined and can play off each other to increase precision, and variations exists with distinct trade-offs in terms of precision, cost of analysis, difficulty of implementation, etc.

The program's approximated state is computed using simplified semantics for control flow: branching is evaluated by taking an upper bound of abstract states when control re-converges. Iteration is treated by taking a computable approximation of the least fixpoint of the body of the iteration, which may be non-trivial. The combination of those lattices and simplified semantics give the technique its name.

As the complexity and/or cost of a program is a function of its inputs, it is natural to try to obtain safe approximations of that cost from safe approximations of its inputs. This is the path taken by the *CiaoPP*[4] static analysis software, built on top of the *Ciao Prolog*[5][31] general-purpose logic programming language. Both are developed by the *CLIP Lab*[6].

The use of abstract domains for resource analysis will be discussed next section. For now, let us focus on the range of programming paradigms and semantics that this analysis setup enables. Computing the abstract domain for a particular aspect of a program boils down to performing a certain computation in the category of abstract domains, which becomes purely a problem in order theory. Built on this base, CiaoPP translates programs into a logic-programming intermediate representation, on which is implemented the analysis itself. As such, *CiaoPP* enjoys a wide range of applications. Over this analysis tool, size and resource analyses have been built for Java source[57], JVM bytecode[63], Logic programming [72], and LLVM-IR[59].

2.10 Recurrence relations

Atop abstract interpretation, a corpus of work dedicated to resource analysis builds parameterized complexities using extraction of *recurrence relations* from source programs. Those recurrence relations implicitly describe sizes/costs/complexities/etc., which then require a solver to be compiled into closed form. The *Costa* analysis framework is based on this principle, and provides both a cost and a termination analysis (see [2]) for a tutorial presentation of the methods). *Costa* has been used to derive safe bounds for time and memory costs in bytecode-style languages, from Java source, the ABS modeling language for concurrent systems[3], and Ethereum smart contracts[10].

[4] https://github.com/ciao-lang/ciaopp
[5] https://ciao-lang.org
[6] https://www.cliplab.org, *The Computational logic, Languages, Implementation, and Parallelism Laboratory*

At the core of *Costa* and related projects is the concept of a recurrence relation (or cost relation)[9]. Those are partial, multi-valued functions in $\mathbb{N}^k \to \mathbb{N}$ defined by part. Namely, given arguments $\vec{x}$, each part of the function is a pair $\langle e, \varphi \rangle$, where e is some expression with free variables $\vec{x}$, algebraic operations, functions *max*, *log*, and *pow*, and function variables $\tilde{c}$. This expression is guarded by φ, a set of linear inequalities over the $\vec{x}$. The pair $\langle e, \varphi \rangle$ defines a partial function from $\{\vec{x} \in \mathbb{N}^k \mid \varphi(\vec{x})\}$ to $\mathbb{N}$ whose value is e. A recurrence relation is then a set of parts which defines a partial, multi-valued function $\tilde{c}$, and a recurrence system is a set of mutually-defined recurrence relations.

The analysis proceeds by generating from a control flow graph a recurrence system, then instantiating of that system. This enables both termination and cost analysis: when the cost metric is time, any instance of that system is a witness of termination, and dually, every proof of termination obtained by instantiating a recurrence system gives a cost bound. Nevertheless, finding good bounds requires not only providing a witness, but trying to find a minimal or small one, which is a significant area of study in that branch of resource analysis[4]. Given the heuristic nature of the instantiation algorithms used and the flexibility of recurrence systems, it is hard to give a firm class of programs, complexities, and languages it fully treats.

Memory bounding in Costa benefits from advanced support for garbage-collection aware memory costs, allowing for different GC strategies to be taken into account to obtain realistic bounds (GC runs when exiting scope, GC runs ideally, GC runs when the heap is full, etc.)[11]. When aliasing and mutability of heap structures are added into the mix, *Costa* can rely on abstract interpretation to infer reachability and acyclicity information. Otherwise, it is restricted to programs where heap-allocated mutable variables are demoted to local variables on which analysis can proceed normally. This is the approach we will take, compiling mutability in imperative blocks to pure functional code. While *Costa* targets languages where mutability may be pervasive, we limit ourselves to uses we know to be analyzable.

2.11 Insights from previous work

Having taken a tour of significant advances in resource analyses for λ-calculus based languages, some points appear as salient in our search for a reusable formalism, and some pain points reveal themselves as obstacles to its implementation.

Insights As we have seen in the beginning of this section, the problem of resource analysis can be reduced, without loss of generality, to one of finding a potential function from program states to resources satisfying some conditions. When programs are typed and functional, and evaluation implemented by substitution, those conditions can be understood as *static*, that is derivable from source code, and *local*, that is associated to individual nodes in the reduction of programs. This allows the creation of a wide-ranging spectrum of resource analyses.

Furthermore, the work by M. Hoffmann and J. Hoffmann shows the theoretical possibility and feasibility of this method, as to provide typing-based resource analyses. Inferring tight potential functions requires not only reasoning on resources at type-level, but on also sizes and number of patterns in data structures, as to derive algorithmic invariants. The index system developed for AARA allows polynomial complexities to be inferred by revealing the underlying uniform iterations in recursive programs over polynomial data-types.

While this body of work is itself impressive, efforts made to extend indexing systems and rephrase them using more mainstream approaches have allowed constraint-based type-system to be defined, as to encode the potential method. dℓPCFand λ_{amor}make use of linear types and (co)monadic programming to separate the concerns of resource preservation, ordering of reductions in time, assignment of indices to types, and instantiations of those indices to derive bounds.

This shows the possibilities of implementing more general resource analyses in a way that separates concerns for compilation, typing, and constraint solving. This would not only make for simpler implementations, but would also be a boon for further work, which would only need to replace parts of the analyzer to change the scope or precision of the analysis. Given the still evolving status of the field of research, such a "lab bench" for resource analysis in general, and AARA analyses in particular, would be quite welcome.

Avenues for improvement Nevertheless, it is not obvious that such expectations can be met as-is. While formalisms in the style of dℓPCFhave been built upon, they remain unimplemented. J. Hoffmann's implementation of AARA only support the polynomial index system presented in this section, while Hoffmann itself developed finer ones covering more flexible types and finer complexities. There is no easy way to fill the gap between theory and implementation: as shown by dℓPCF, index systems may grow so much as to be intractable.

Furthermore, all the systems shown above focus on a core language whose fixed semantics are a limitation to future extensions. RAML operates only on call-by-value λ-calculus, and both dℓPCFand λ_{amor}have separate type systems for call-by-value and call-by-name strategies. If one wants to implement an effect system to recover some form of mutable state or exceptions (to give two example), one must significantly extend the semantics of the base λ-calculi. Indeed, implementing monads as a library for call-by-value evaluations introduces higher-order functions and the accompanying indices in trove, and ensuring those do not increase the complexity of the index instantiation procedure is not trivial.

Our proposed solution In this thesis, we implement a novel resource analyzer following the AARA method, which addresses those issues. First, the index system is implemented as a constraint-based extension to a mainstream linear type system. Erasing the indices, we recover established formalisms from type theory and proof theory, which opens the door to integrating techniques and implementation from a wide-ranging corpus. The constraints we use are a sub-language of first-order logic, and we interface off-the-shelf solvers to reuse as much as possible standard algorithms in constraint solving. It also makes explicit the link between AARA's index system, dℓPCFrelative inference procedure which instantiates indices as function symbols, and Herbrand's elimination technique[30] for quantifier elimination. This will give a thorough account of the privileged position polynomials complexities occupy in AARA.

Secondly, we define and implement our analysis in a *call-by-push-value* abstract machine. This setup allows us to immediately implement simultaneous call-by-name and call-by-value strategies, and to mix-and-match the two in source programs. We show how the set of primitives used by λ_{amor}can be significantly compacted using the relation between *call-by-push-value* and (co)monadic programming. This allows us to reintroduce M. Hoffmann's original linear tokens into a modern formalism, closing an important loop. This formalism also allows monadic effects to be supported at no cost for the analyser: source programs have a *do-notation* for nested effects (mutable state and exceptions), compiled into the abstract machine without a need for special primitives. Another significant technique in the implementation of full functional languages is available to encode source programs more faithfully: first class and defunctionalized continuations. Those allow for a first-class representation of computation context which greatly

simplify the resource-aware typing of functions, and allow for computations with custom calling conventions.

Plan of the rest of the manuscript We introduce the abstract machine we'll use as the core of our analysis in the next sections (3 and 4). Its runtime semantics are an instance of the *Call-by-push-value* paradigm, which refines evaluation strategies for λ-calculi. We will justify this choice, then introduce a simple type system for the machine. This type system will turn out to be exactly *polarized intuitionistic linear logic*, and we shall present the significance of that fact for the theoretical backing of our framework. We then show how source languages, independently of resource analysis, are compiled into the machine (5). This frontend compilation step expands the monadic do-notation in the source language into monadic primitives for mutable state and exceptions, and we will discuss how the call-by-push-value formalism enables those primitives to be safely elided from the intermediate representation without complex algorithm reasoning on the monadic laws.

Our extensions for static analysis will be developed in section 6. Namely, we will present our index system over the machine, in line with λ_{amor}. Our system will diverge from the previous work, as it enables a simpler representation of resource that subsumes λ_{amor}, and admits automatic elaboration of resource manipulations, allowing source programs to automatically be made resource aware. From that point onward, we will be able to present our resource analysis for functional languages in 7. It takes the form of a compilation scheme to-and-from the abstract machine. We will explain how the machine allows for separation of concerns within the analysis, and how to enrich datatype definitions to bridge the gap between RAML-style and dℓPCF-style index systems.

Our implementation will be presented in chapter 8. The tooling we implemented will be presented, and our type inference procedure will be detailed. The choice of a constraint-based type inference procedure will be justified as this point. We shall furthermore address post-processing and solving the constraints created during type inference. This will feature a discussion on quantifier elimination which will enlighten the favored status of polynomials indices in AARA. This presentation will end with a presentation of how SMT solvers interface with our code.

Armed with a proved and implemented analysis, we will be able to compare our work to the state of the art (chapter 9). The main evaluation will be against RAML. We shall also comment on the limitation of our work, both the theory and implementation, which will offer possible avenues for further work. Our contributions in terms of re-usability, extension in scope and compatibility with external tools will be explicated.

Finally, we will be able to suggest avenues for further work and summarize the insights brought on by this manuscript, which will allow us to open up the discussion beyond static analysis in our final words (10).

Chapter 3

Abstract machines for operational semantics

In this chapter, we introduce the main principle backing our novel analysis: analyzing abstract machines with good properties, which allows resources (de)allocation to be modeled as an effect in a suitable type system. We present the non-deterministic, call-by-value (CBV), and call-by-name (CBN) semantics of simply-typed lambda calculus, and comment on the problem they pose for resource analysis of functional languages. Abstract machines for those semantics are then introduced, foreshadowing our formalism.

3.1 λ-terms and operational semantics

Full λ-calculus Recall that we introduced amortized analysis as a transition system $(\mathcal{C}, \triangleright)$ to which each transition $c \triangleright c'$ is associated a cost $\mu(c, c')$ by a cost metric μ. The total costs of a program is the largest partial sum of cost attained during reductions. Under this setup, $\triangleright$ represents the *small-step* semantics of a programming language. This is a problem for languages based on λ-calculi. To make things formal, we introduce λ-*terms* t, values V and *contexts* $C[]$ in figure 3.1. Contexts are as usual λ-terms with a single hole $[]$, and values are (for now) all terms. We add pairs to the language, equipped with both pattern-matching $\mathbf{let}\,(x, y) = t\,\mathbf{in}\,t$ and projections $\pi_1(t)$ and $\pi_2(t)$. Those will come useful later.

We write $C[t]$ for the λ-term generated by replacing the only instance of $[]$ in the context $C[]$ by the term t. We define β-reduction under context as the closure under context of basic

head-reductions. Formally, the small step reductions are:

$$\begin{aligned}(\lambda x.t)\, V &\triangleright t[V/x] \\ \mathbf{let}\,(x,y) = (V,V')\,\mathbf{in}\,t &\triangleright t[V/x, V'/y] \\ \pi_i((V_1,V_2)) &\triangleright V_i \\ C[t] &\triangleright C[u] \quad \text{whenever } t \triangleright u\end{aligned}$$

This relation is immediately not deterministic. For example, in the term $(\lambda x.x)((\lambda y.y)z)$, both x and y can be substituted. Nevertheless, two reductions sequence for a term will always converge to the same value. This is unfortunately not a strong enough property for our purposes: there is no easy way to compute, given a given cost metric for the λ-calculus, which of the possible reduction sequence induces the largest footprint. Furthermore, complete reduction (within a function's body for example) may not be desirable depending on the programming language being modeled.

To solve this problem, we can limit reduction as to give priority to only one possible reduction within a term, and to never reduce under a λ (this later requirement is known as *weak-head* reduction). This is done with an *evaluation strategy* The most common two are *call-by-value*(CBV) and *call-by-name*(CBN).

Call-by-value In CBV evaluation, when two reduction rules can be applied to a term, the *deepest and right-most* one is picked. This can be achieved by keeping the previous notions of terms and β-reductions, and introducing restricted notions of contexts and values. Formally, the syntax is as in figure 3.2.

With those *by-value* contexts, it is now impossible to reduce either, continuation within a **let**...**in**, or abstractions before their arguments are fully evaluated. This guarantees that

$$\begin{aligned}t &::= x \mid t\,u \mid \lambda x.t \mid (t,u) \mid \mathbf{let}\,(x,y) = t\,\mathbf{in}\,t \mid \pi_{i\in\{1,2\}}(t) \\ V &::= t \\ C[] &::= [] \mid C[]\,t \mid t\,C[] \mid \lambda x.C[] \mid (C[],t) \mid (t,C[]) \\ &\quad \mid \mathbf{let}\,(x,y) = C[]\,\mathbf{in}\,t \mid \mathbf{let}\,(x,y) = t\,\mathbf{in}\,C[] \mid \pi_i(C[])\end{aligned}$$

Figure 3.1: Simply-typed λ-calculus

$$
\begin{aligned}
t &::= x \mid t\,u \mid \lambda x.t \mid (t,u) \mid \mathbf{let}\,(x,y) = t\,\mathbf{in}\,t \\
V &::= x \mid \lambda x.t \mid (V,V') \\
C[] &::= [] \mid C[]\,V \mid t\,C[] \mid (C[],V) \mid (t,C[]) \\
&\quad \mid \mathbf{let}\,(x,y) = C[]\,\mathbf{in}\,t \mid \mathbf{let}\,(x,y) = V\,\mathbf{in}\,C[]
\end{aligned}
$$

Figure 3.2: Call-by-Value λ-calculus

all substitutions only reduce the *down-most, right-most* term that isn't under a λ, and it can be shown (we shall omit the proof of this standard result) that the *by-value* β-reduction hereby defined is indeed deterministic. This evaluation strategy is used by ML-style languages, JavaScript, Scheme, etc.

Pattern-matching pairs The CBV λ-calculus we introduced does not include the two projection primitives for pairs. When building a pair (t_1, t_2) in the CBV semantics, both components must already be evaluated, giving a value (V_1, V_2), and the binder $\mathbf{let}\,(x_1, x_2) = (V_1, V_2)\,\mathbf{in}\,t$ merely binds *both* components of the pair, leading to a term $t[V_1/x_1, V_2/x_2]$. When x_i is not free in t, the component t_i of the pair must still be evaluated to some value V_i, as $[t_i/x_i]$ is not a valid substitution (we may only substitute values). Therefore, the syntax we chose for pairs imposes the CBV semantics on them.

Call-by-name Other restrictions to contexts and values lead to other deterministic β-reductions. Of importance here, *call-by-name* reduction is the "dual" of call-by-value: it reduces the bodies of applications before their arguments, the later being only reduced after substitution. This is achieved with the choice term, context, and values below. This time, we keep the projection pair and remove pattern-matching pair, as seen in figure 3.3.

$$
\begin{aligned}
t &::= x \mid t\,u \mid \lambda x.t \mid (t,t) \mid \pi_{i\in\{1,2\}}(t) \\
V &::= x \mid \lambda x.t \mid (t,t) \\
C[] &::= [] \mid \lambda x.C[] \mid C[]\,t \mid (C[],t) \mid (t,C[]) \mid \pi_i(C[])
\end{aligned}
$$

Figure 3.3: Call-by-Name λ-calculus

Many more possible evaluation strategy exists, but those two have important benefits: if reducing a term t terminates with any strategy, then reducing it in CBN also terminates, and if the CBV reduction of t terminates, then it terminates faster than all other evaluation strategies. In that sense, CBV is the *fastest* strategy, and CBV is the *safest*.

Projection pairs With CBN semantics, all expressions (t_1, t_2) are already values, and the projection $\pi_i((t_1, t_2))$ reduces to t_i, which itself reduces later to some V_i. If the other component t_j (with $i \neq j$) of the pair is not projected, it is *never* evaluated. Once again, the syntax we chose for pairs enforces CBN semantics.

Past λ-calculus To encode advanced features of functional programming languages, both evaluation strategies should be used. For example, an iterator on a data structure naturally involves both a lazy thunk (the computation done at each step of iteration) and a value (the thing being iterated over). Iterators should ideally be composable without overhead, and be first-class values. But if the computation is encoded in CBV, extraneous closures are involved and cannot be all optimized away. On the other hand, if the value being iterated over is a thunk, the evaluation of the iterator will interleave iteration steps and generation of new data structures node. This interleaving may increase the footprint of the program, and rarely matches programmer intent when using an iterator. Neither pure CBV not CBN can encode first-class iterators as desired. As such, we wish for a formalism that can mix CBV and CBN evaluation. This requires combining the two conflicting notions of values. Abstract machines will provide this unification with *call-by-push-value* semantics. First, let us present the abstract machine account of CBV and CBN evaluation, as to slowly introduce notation and concepts.

3.2 Abstract machines

As far as providing a systematically-reusable formalisms for resource analyses is concerned, we find that using abstract machines as opposed λ-calculi provides a satisfying starting point. To give an first idea of why this is the case, consider the reduction under context $C[t] \triangleright C[t']$, with $t \triangleright t'$ the head reduction being considered. It has to be assumed, in the general case, that this reduction manipulates resources, and that this may involve a transfer of resource between the context $C[]$ and the sub-term t being reduced. Therefore, if the reduction $t \triangleright t'$ releases

resources from the term, where do they end up? Are they released ? If they are transferred, then where exactly ? Correspondingly, if $t \triangleright t'$ consumes resources, where are those taken from ? If taken from somewhere in the context, λ-calculus gives no information as to the origin of those resources. Formally, it is far from obvious how to relate the *local* cost $k(t, t')$ and the *global* cost $k(C[t], C[t'])$. This means that reasoning about resources cannot be done locally without the aid of a machinery relating local and global costs. Abstract machines provide a systematic answer to those questions: machines only involve head-reduction at the root of the program, doing away with the notion of a surrounding context. Furthermore, we shall see in later chapters that we can encode resource manipulations in such a way that resources stay at a bounded depth we control, never being assigned to sub-terms "deep" inside the machine. This will allow us to combine the simplicity of local cost expressions and the flexibility is global costs.

Generalities and notation Abstract machines encode λ-terms has a pairing of a *term* t and a *environment* e, written $\langle t \parallel e \rangle$ and a called a *command*. This is a continuation-passing style formalism: t is the "thing currently being evaluated" and e is its continuation. Alternatively, e is the "rest of the program" and t its antecedent. Those terms and environments do not to map directly to λ-terms and contexts.

Amongst terms, some are *weak values* $V \subset t$. We use the wording "weak" values to emphasize that values need not be fully evaluated, only sufficiently determined, in the same way that weak head reduction need not to completely evaluate a term. By symmetry, sufficient determined environments are *weak stacks* $S \subset e$. We shall use the terms "values" and "stacks" without the adjective "weak" for convenience. The key principle of abstract machines is that when a value interacts with a non-stack environment, control is taken by this environment. Likewise, stacks interacting with non-value terms wield control to the term.

Just as λ calculus is first a non-deterministic term-rewriting system, on which some restrictions can be applied to create determinism, the first abstract machine we'll see is non-deterministic. First, we give a core syntax of command, terms, environments, values and stacks. We shall only introduce functions and pairs later, as to focus on the basic syntax and control structures the machine uses. This core grammar is given in figure 3.4.

$c ::= \langle t \parallel e \rangle$	$\langle V \parallel \mu x.c \rangle \rhd^{\beta} c[V/x]$	**(bind-V)**
$t ::= V$	$\langle \mu a.c \parallel S \rangle \rhd^{\beta} c[S/a]$	**(bind-S)**
$V ::= x \mid \mu a.c$		
$e ::= S$	$\mu x.\langle x \parallel e \rangle \rhd^{e} e$	**(eta-bind-V)**
$S ::= a \mid \mu x.c$	$\mu a.\langle t \parallel a \rangle \rhd^{t} t$	**(eta-bind-S)**

Figure 3.4: Abstract machines: control structures

Command c are always a term t interacting with an environment e, written $\langle t \parallel e \rangle$. Values contains variables x, y, z and stack contain stack variables a, b, c. On each side, a binder denoted by μ substitutes the other side of a command for a value/stack-variable in another command: a value $\mu a.c$ substitutes a stack for a in c, and a stack $\mu x.c'$ substitutes a value for x in c'. We give two reductions **(bind-V)** and **(bind-S)** which are β-rules, written $\rhd$. We also define two eta-reduction, one for terms and one for environments, which eliminate redundant μ binders. Those arewritten $\rhd^t$ and $\rhd^e$ respectively. They aren't required for evaluation, only to provide normal forms to terms and environments within the theory.

The two β-rules *do not apply under context*. In fact, no notion of "context" has been defined for commands, and none will be in this work. This means all reductions happens on the root command of the machine. Since there is no context, all transfer of resource we may define on the machine will happen between the left (value) side and the (right) stack side. In the general case, reductions $c \rhd c'$ always have an *active* side in which c' occurs as a sub-command of c, and a passive side which ends up being substituted for a variable in c'. Weak values and weak stack are always passive in this situation. When adding resources to the machine in the next chapter, resource shall always flow from the passive to the active side.

So far, the machine is non-deterministic. The only critical pair in the machine (i.e. the only command which has more than one possible reduction), is the command $\langle \mu a.c_1 \parallel \mu x.c_2 \rangle$. Depending on the rule being used, the active side may be on the left or right. Determinizing this ambiguity leads to a machine with CBV or CBN semantics, as we will see later in this section. For now, let us briefly introduce some data structures to the machine. We shall introduce some type constructors at the same time, while keeping the formal introduction of the type system for the machine for the next chapter. Type expressions are written in capital letters $(A, B, C, \dots)$.

Pairs The abstract machine formalism enables us to separate projection pairs and pattern-matching pairs into different types. The case of pattern-matching pairs to the machine is rather straightforward: given two values V, V', the pair (V, V') is a value of type $A \otimes B$. On the right side, the binder $\mu x.c$ is extended to bind the two components of a pair, giving a stack $\mu(x, y).c$. The reduction rule $\langle (V, V') \parallel \mu(x, y).c \rangle \triangleright c[V/x, V'/y]$ merely unpacks a pair into a new command c. Note that when this reduction occurs, execution proceeds to c, which is on the right side of the original command: the right side is active, the left side is passive.

As for also are projections pairs, we use two projections π_1 and π_2 that can be *pushed on a stack* S, giving two new stacks $\pi_1 \cdot S$ and $\pi_2 \cdot S$, of type $A \& B$. A projection pair value of the same type then pops a projection π_i from the stack and yields control to a command that eventually returns the projected component of the pair to rest of the stack S. Formally, the $\mu a.c$ binder is adapted to perform case analysis on the projection on top of the stack, the same way pattern-matching would inspect a value and branch. The syntax is $\mu(\pi_1(a_1).c_1, \pi_2(a_1).c_2)$: this value projects to its first component by matching on a projection π_i, binding a continuation to a_i, and yielding control to c_i. Reduction is then defined as:

$$\langle \mu(\pi_1(a_1).c_1, \pi_2(a_2).c_2) \parallel \pi_i \cdot S \rangle \triangleright c_i[S/a_i]$$

The syntactic additions and rules for those pairs and corresponding rules are summarized below:

$$\begin{aligned} V &::= (V, V') \mid \mu(\pi_1(a).c, \pi_2(a').c') \mid \dots \\ S &::= \mu(x, y).c \mid \pi_1 \cdot S \mid \pi_2 \cdot S \mid \dots \end{aligned}$$

$$\begin{aligned} \langle (V_1, V_2) \parallel \mu(x_1, x_2).c \rangle &\triangleright^\beta c[V_1/x_1, V_2/x_2] && \textbf{(match-pair)} \\ \langle \mu(\pi_1(a_1).c_1, \pi_2(a_2).c_2) \parallel \pi_i \cdot S \rangle &\triangleright^\beta c_i[S/a_i] && \textbf{(proj-pair)} \end{aligned}$$

$$\begin{aligned} \mu(x_1, x_2).\langle x_1 \otimes x_2 \parallel e \rangle &\triangleright^e e && \textbf{(eta-match-pair)} \\ \mu(\pi_1(a).\langle t \parallel \pi_1 \cdot a \rangle, \pi_2(a_2).\langle t \parallel \pi_2 \cdot a_2 \rangle) &\triangleright^t t && \textbf{(eta-proj-pair)} \end{aligned}$$

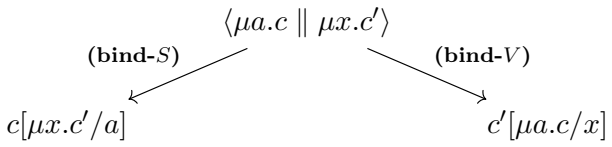
Functions Functions are encoded in continuation-passing style: a function call is made up of a function, an argument, and a stack representing the continuation on the call. They have type $A \multimap B$. The argument/continuation pair is stored on the right side by pushing the argument on the stack: we write $V \cdot S$ for the argument V pushed onto the stack S. On

the other side, a function is made of a binder that simultaneously binds the argument and continuation, and jumps to a new command encoding the body of the function. As such, functions value are denoted $\mu(x;a).c$, where x is the name of the formal argument, a is the continuation variable, and c the body of the function. The reduction again unpacks one side of the machine (the $V \cdot S$ on the right side) into the other side. We therefore have a reduction rule $\langle \mu(x;a).c \parallel V \cdot S \rangle \triangleright c[V/x, S/a]$. This is summarized by the following addition to the grammar and rules of the machine.

$$V ::= \mu(x;a).c \mid \dots$$
$$S ::= V \cdot S \mid \dots$$

$$\begin{array}{ll} \langle \mu(x;a).c \parallel V \cdot S \rangle \triangleright^{\beta} c[V/x, S/a] & \textbf{(fun)} \\ \mu(x;a).\langle f \parallel x \cdot a \rangle \triangleright^{t} t & \textbf{(eta-fun)} \end{array}$$

This machine is non-deterministic, but its reduction is much more manageable than the corresponding λ-calculus. Indeed, there is only one critical pair of reductions which apply to the same command. For example, the command $\langle \mu a.c \parallel \mu(x \otimes y).c' \rangle$ can only reduce with **(bind-S)**, as the left side isn't some (V, V'). The only ambiguous pair is given below:



To create a deterministic machine, it is only necessary to resolve this single conflict. By making the **(bind-S)** rule apply, the evaluation of terms dominates the evaluation of their surrounding environments, giving the machine CBV semantics. Making the other choice gives the machine CBN semantics where terms are only evaluated when their environment can no longer reduce without this evaluation.

3.3 Interlude: Focusing and deterministic machines (1/2)

The creation of deterministic machines from the non-deterministic one follows the principle of *polarization*, which comes from the study of linear logic. Linear logic is a formal logic system introduced by Girard in the 80's[13, 27, 73], whose *polarized* extension sheds lights on CBV and CBN evaluation.

Focused linear logic The abstract machine we have presented is a direct incarnation, as a programming language, of a logical framework called *focused intuitionistic linear logic*. This will be made explicit next chapter. When presented in sequent-style, focused intuitionistic linear logic is a set of rules relating a *sequent* $\Gamma \vdash \Delta$ called the *conclusion* to zero, one, or many other sequents called the *premises*. We focus on a subset of the system for this interlude. A proof in linear logic is a tree whose nodes are all applications of rules, and each edge relates the conclusion sequent of a rule to the premise sequent of another. The sequents themselves are made up of a finite set of formulas Γ, and a single formula Δ, and each formula follows the grammar

$$A ::= X \mid A \otimes A \mid \mathbb{1} \mid A \parr A \mid \top.$$

All rules of linear logic remove one of the connectives in this grammar from its conclusion, to form smaller premises. This is the *sub-formula property*. Programmatically, this corresponds to building up and breaking down data within terms and environments. The only exception to this principle is a rule called "cut" shown below. In some sense, "cut" links up a proof producing some data A on the left and one consuming it on the right. We say that the proof *cuts on* A. Programmatically, this is the logical twin to our commands, and bring compatible sub-programs together to interact.

$$\frac{\Gamma \vdash A \qquad \Gamma', A \vdash \Delta}{\Gamma, \Gamma' \vdash \Delta}\ (\text{cut})$$

Cut-elimination We shall not delve too long on linear logic itself here, but do mention some important properties here to justify the approach. First, let us define the sub-formula property properly. Let A, B be formulas. We say B is a sub-formula of A if its syntax tree is a sub-tree of the syntax tree of A, and call it a strict sub-formula if furthermore $A \neq B$. This is extended to sequents by stating that a sequent $\Gamma \vdash \Delta$ is a sub-sequent of $\Gamma' \vdash \Delta'$ if all formulas in $/Gamma \cup \Delta$ are a sub-formula of a formula in $\Gamma' \cup \Delta'$. A sub-sequent is strict if at least one of its formula is a strict sub-formula. Finally, a proof has the sub-formula property if for all rules in it, the premises are all strict sub-sequent of the conclusion. Figuratively, this means every intermediate step of such proof involves "easier/simpler" sequents that the root one. In linear logics, all rule except *(cut)* have the sub-formula property, every proof can be transformed into one with the sub-formula property by removing all instances of the *(cut)* rules, and there is an *algorithm* implementing this transformation called *cut-elimination*.

Proofs as programs This *cut-elimination* algorithm is a rewriting on proofs, which replaces a *cut* chosen non-deterministically with zero-to-many cuts further away from its root. Iterating this cut-elimination is proven to terminate and to result in a proof without cuts. This is an execution model for proofs: the starting proof is program which is executed by successively rewriting it, removing a link between data production and data consumption, which eventually terminates. Programmatially, cut-elimination is execution.

Compare this with typed λ-calculi. A syntax tree is reduced by successively rewritten (through β-reduction), which substitutes data having been created to a place where it is consumed. Cut-elimination, as we describe it, is non-deterministic, but is made deterministic by introducing *polarity*.

Polarization Polarization is a technique allowing a simpler proof system to be derived from linear logic. The resulting system has a straightforward *erasure* translation to the original linear logic, but is simpler in the sense that many proof which are "morally" identical but have different cut-elimination behavior are removed. This is useful to use since, by the proofs as program paradigm, those correspond to a simplification in the runtime behavior of programs, while keeping all programs expressible. The principal tool of polarization is a partition of formulas A into positive formulas A^+ and negative formulas A^-, according to the following grammar:

$$\begin{aligned} A &::= A^+ \mid A^- \\ A^+ &::= X^+ \mid A^+ \otimes A^+ \mid \mathbb{1} \mid \Downarrow A^- \\ A^- &::= X^- \mid A^- \parr A^- \mid \top \mid \Uparrow A^+ \end{aligned}$$

Polarities divide the formulas of linear logic according to their root connectives. This partition of formulas gives proofs important properties regarding cut-elimination, which translate to abstract machines. Namely, when all significant (in some way) cuts in a proof are cuts on positive formulas, this proof can be simplified by breaking down the premises into either of X^+, $\mathbb{1}$, or $\Downarrow A^-$ (an "inverted" premise), then cutting one of those premises on with cut-free proof (we say the proof *focuses* on this premise), and finally recursing on the remaining premise. This has a straightforward translation into operational semantics. Indeed, a cut-free premise is a fully-reduced "value", and a cut below it passes this value to a yet-to-be-reduced continuation.

This is the operational definition of CBV: control is passed to a continuation iff the term being passed to it is in normal form.

A similar result exists on proofs which only cut on negative formulas: conclusions can be iteratively inverted and cut along a negative formula, whose other premise is cut-free. This gives an equivalent proof-as-program where no continuation is provided a term until that continuation is fully reduced: this is CBN.

Inversion The restriction on polarity of cuts applies only to *significant* cuts, that is to say to cuts of *inverted* sequent. Inverting a sequent means applying all rules of the system to it as long as those rules are *reversible* (and therefore irrelevant, since they can be cancelled later on). Inverting a sequent, while a pretty abstractly-motivated transform at first, as an immediate translation to λ-calculus and functional languages. Indeed, consider as an example a CBV program in which a pair is created each component matched. We can, for example, write the following two programs:

```
...                       ...
let z = e1 in             let z = e1 in
let t = e2 in             match (x,y) = z in
match (x,y) = z in        let t = e2 in
...                       ...
```

Those two program fragments are equivalent at runtime: one can evaluate `e2` before or after matching the components of the pair. This is what inversion does: it transforms the left one into the left one, using the fact that the already evaluated pair `z` and two variables `x` and `y` are, for all intent and purposes, the same thing. Inversion can therefore move the pattern-matching as early as possible without changing semantics, giving simpler programs.

This gives the key to determinizing our abstract machines. Our non-deterministic abstract machine corresponds to a subset of linear logic, which can be made to have CBV-like semantics by making all relevant cut be on positive formulas, or CBN-like by making all relevant cuts occur on negative formulas. This is exactly what we shall do by introducing first a CBV and CBN machine, then a focused one combining both.

3.4 Call-by-value machine

We annotate the syntax of the CBV machine with an exponent "+" to denote its *positive* restriction, and only change the syntax of the two binders involved in the critical pair. Continuation-capture is promoted to a full term, which we highlight so in in the grammar. On the other hand, the dual binder $\mu x.c$ is not promoted, and remains a stack. This is emphasized so:

$$\begin{aligned} c &::= \langle t^+ \parallel e^+ \rangle \\ t^+ &::= \mu a^+.c \mid V^+ \\ V^+ &::= x^+ \mid \mu(x^+; a^+).c \mid (V^+, V^+) \\ e^+ &::= S^+ \\ S^+ &::= a^+ \mid \mu x^+.c \mid V^+ \cdot S^+ \mid \mu(x^+, x^+).c \end{aligned}$$

Let us comment and explain this change. In CBV λ-calculus, data is passed around in function calls, and CBV ensures functions cannot consume their arguments unless the latter is fully evaluated. Making $\mu a^+.c$ a non-value prevents the $\langle V^+ \parallel \mu x^+.c \rangle$ from applying to it, resolving the critical pair in favor of **(bind-S)**. Then, given a continuation-capturing term in a command like $\langle \mu a^+.c \parallel e^+ \rangle$, the machine always ends up evaluating $c[e^+/a^+]$. This means the command evaluating the value-side are always run first, and that all reductions of e^+ are blocked until they bubble back up to the root of the program. This evaluation strategy is exactly the inversion/focusing process seen above. In this system, any command involving $\mu(x, y).c$ is invertible, but $\mu(x; a).c$ is not.

Pairs are provided as pattern-matching pairs: a term $\mu a^+.c$ must evaluate to a pair by reducing to $\mu a^+.\langle (V_1^+, V_2^+) \parallel a^+ \rangle$ before the continuation $\mu(x,^+ y)^+.c$ is able to match on the pair to binds its two components. On the other hand, the term may at any time bind the entire environment to a^+, delaying its evaluation.

Next, consider the function $f = \mu(x; a).c_f$, being called with an argument t (which evaluates to V) and a continuation b. We omit the exponent for brevity. This function call is written:

$$c = \langle t \parallel \mu y.\langle \mu(x; a).c_f \parallel y \cdot b \rangle \rangle.$$

The grammar imposes that t is either $\mu d.c_t$, or a value V. If it is a value, then its evaluation is over, and the only applicable rule for c is **(bind-V)**, giving the function call $\langle \mu(x;a).c_f \parallel V \cdot b \rangle$. Otherwise, the current state is $c = \langle \mu d.c_t \parallel \mu y.\langle \mu(x;a).c_f \parallel y \cdot b \rangle \rangle$, and the only applicable rule is **(bind-S)**. In this case, the entire right side representing the call is substituted into c_t, and reduction proceeds by evaluating the term, only now with an explicit continuation instead of the variable d. In both cases, there is no way the function can capture its argument before the latter is done evaluating.

Thanks for focusing and inverting, forcing a CBV evaluation strategy onto the machine doesn't involve creating contexts, merely upgrading a single construct (the continuation-capture $\mu a.c$) from a value to a term. The relative simplicity of this method increases further when adding pairs and functions, as it doesn't require any changes to preserve CBV semantics. Compare this to λ-calculus, where new contexts must be added to preserve CBV semantics when adding pairs (the same doesn't apply for functions, which have special status in λ-calculus). By symmetry, making the dual choice of promotion, we can force CBN semantics.

3.5 Call-by-name abstract machine

Dually to CBV, let us promote the value-binder $\mu x.c$ from a stack to an environment, protecting it from being captured by $\mu a.c$ on the other side. We furthermore add a "$-$" exponent to the CBN terms and environments. This is highlighted in the following grammar:

$$
\begin{aligned}
c &::= \langle t^- \parallel e^- \rangle \\
t^- &::= V^- \\
V^- &::= x^- \mid \mu a^-.c \mid \mu(x^-;a^-).c \mid \mu(\pi_1(a_1^-).c_1, \pi_2(a_2^-)c_2) \\
S^- &::= a^- \mid V^- \cdot S^- \mid \pi_1 \cdot S^- \mid \pi_2 \cdot S^- \\
e^- &::= \mu x^-.c \mid S^-
\end{aligned}
$$

The rules remain unchanged. This means that in the critical pair $\langle \mu a.c \parallel \mu x.c' \rangle$ now only reduces to $c'[\mu a.c/x]$ via **(bind-V)**. In plain language, when evaluating a term $\mu a.c$ with continuation $\mu x.c$, the latter always captures the former. Both function $\mu(x;a).c$ and projection pairs $\mu(\pi_1(a).c_1 \mid \pi_2(a).c)$ are invertible.

As an example of the CBN semantics of the modified machine, let us consider the same function call as in the previous section, $c = \langle t \parallel \mu y.\langle \mu(x;a).c_f \parallel y \cdot b\rangle\rangle$. Whether t is some $\mu a.c$ or not, is it a value, and the machine regardless reduces to $\langle \mu(x;a).c_f \parallel t \cdot b\rangle$ and then to $c_f[t/x, b/a]$. We can see the evaluation of the function call proceeds to the body of the function and that its argument is substituted for the *whole expression* defining its argument, and not its *value*.

3.6 Embedding λ-calculus

The encoding of the simply-typed λ-calculus into the machine is generic, supporting CBV, CBN, and non-deterministic evaluation: the operational semantics is the term is fully decided by the machine. Pairs can be added, with distinct encoding of CBV and CBN pairs.

The two translations from simply-typed λ-calculus is provided below. Each λ-term is translated by a machine term. We write $[\![t]\!]_\varepsilon$ the translation of a term t, with $\varepsilon \in \{+, -\}$. The choice of *polarity* determines the CBV or CBN semantics.

$$
\begin{aligned}
[\![x]\!]_\varepsilon &= x \\
[\![\lambda x.t]\!]_\varepsilon &= \mu(x \cdot a).\langle [\![t]\!]_\varepsilon \parallel a\rangle \\
[\![t\, u]\!]_\varepsilon &= \mu a.\langle [\![t]\!]_\varepsilon \parallel \mu x.\langle [\![u]\!]_\varepsilon \parallel \mu y.\langle x \parallel y \cdot a\rangle\rangle\rangle \\
[\![(t,u)]\!]_+ &= \mu a.\langle [\![t]\!]_+ \parallel \mu x.\langle [\![u]\!]_+ \parallel \mu y.\langle (x,y) \parallel a\rangle\rangle\rangle \\
[\![(t,u)]\!]_- &= \mu(\pi_1(a).\langle [\![t]\!]_- \parallel a\rangle, \pi_2(b).\langle [\![u]\!]_- \parallel b\rangle) \\
[\![\mathbf{let}\,(x,y) = t\,\mathbf{in}\,u]\!]_+ &= \mu a.\langle [\![t]\!]_+ \parallel \mu(x,y).\langle [\![u]\!]_+ \parallel a\rangle\rangle \\
[\![\pi_i(t)]\!]_- &= \mu a.\langle [\![t]\!]_- \parallel \pi_i \cdot a\rangle
\end{aligned}
$$

Variables are translated as is. Abstractions are always values, and are directly translated as function values in the machine. Application is the most involved translation. In CBV semantics, the argument must be evaluated before the body of the function, while the opposite is true in CBN semantics. To achieve this, the application tu, when compiled to the machine, first computes the function value associated to t and binds it to a variable x. Then, the argument expression u is made to interact with a continuation that captures it as y, and evaluates the call. With CBN semantics, this capture is successful and the argument expression

is captured as-is. In CBV semantics, the argument instead captures the continuation. Pairs are translated as either pattern-matching or projections pairs depending on the polarity.

Example Let us consider the function $t = \lambda x.x$ applied to the argument $u = (\lambda y.y)k$, where k is some free variable. First, we compare the reductions imposed by the CBV and CBN semantics in λ-calculus. It shows that under CBV semantics, the application within u is reduced first, then the reduced argument is applied to t. Inversely, in CBN semantics, u is substituted within t first, and only them does the reduction within the argument takes place:

CBV	**CBN**
$t\,u = (\lambda x.x)((\lambda y.y)k)$	$t\,u = (\lambda x.x)((\lambda y.y)k)$
$\to (\lambda x.x)z$	$\to (\lambda y.y)z$
$\to z$	$\to z$

Let us now compile this term into the machine. We write id_x as a shorthand for the identify function $\mu(x \cdot a)\langle x \parallel a\rangle$, where the bound variable x is the argument of the function, repeated in the shorthand for convenience. We shall α-convert the other variables to make them all distinct during translation. With this in mind, let us apply the compilation rules to $t\,u = (\lambda x.x)((\lambda y.y)k)$:

$$\begin{aligned}[\![t\,u]\!] &= \mu a_1.\langle [\![t]\!] \parallel \mu z_1.\langle [\![u]\!] \parallel \mu z_2.\langle z_1 \parallel z_2 \cdot a_1\rangle\rangle\rangle \\ &= \mu a_1.\langle \mathrm{id}_x \parallel \mu z_1.\langle \mu a_3.\langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot a_3\rangle\rangle\rangle \parallel \mu z_2.\langle z_1 \parallel z_2 \cdot a_1\rangle\rangle\rangle\end{aligned}$$

To evaluate this translated term, we make it interact with a fresh continuation variable $\star$ (which is convention when continuation thought of as final are added to the machine). This will eventually reduce to some $\langle V \parallel \star\rangle$. We highlight the head command in red at each step to facilitate reading. The beginning of the reduction is identical for both CBN and CBV machines:

$$\begin{aligned}&\langle \mu a_1.\langle \mathrm{id}_x \parallel \mu z_1.\langle \mu a_3.\langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot a_3\rangle\rangle\rangle \parallel \mu z_2.\langle z_1 \parallel z_2 \cdot a_1\rangle\rangle\rangle \parallel \star\rangle \\ \triangleright\quad &\langle \mathrm{id}_x \parallel \mu z_1.\langle \mu a_3.\langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot a_3\rangle\rangle\rangle \parallel \mu z_2.\langle z_1 \parallel z_2 \cdot \star\rangle\rangle\rangle \\ \triangleright\quad &\langle \mu a_3.\langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot a_3\rangle\rangle\rangle \parallel \mu z_2.\langle \mathrm{id}_x \parallel z_2 \cdot \star\rangle\rangle\end{aligned}$$

At this point, we end up in a command $\langle \mu a_3.\cdots \parallel \mu z_2.\cdots\rangle$ with behaves differently in CBV and CBN semantics. Let us follow the CBV reduction, in which the μa_3 binder has priority,

and write $\triangleright^+$ for the CBV reduction. Starting from the last command, the reduction is:

$$
\begin{aligned}
& \langle \mu a_3.\langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot a_3\rangle\rangle\rangle \parallel \mu z_2.\langle \mathrm{id}_x \parallel z_2 \cdot \star\rangle\rangle \\
\triangleright^+ \quad & \langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot \mu z_2.\langle \mathrm{id}_x \parallel z_2 \cdot \star\rangle\rangle\rangle\rangle \\
\triangleright^+ \quad & \langle k \parallel \mu z_4.\langle \mathrm{id}_y \parallel z_4 \cdot \mu z_2.\langle \mathrm{id}_x \parallel z_2 \cdot \star\rangle\rangle\rangle \\
\triangleright^+ \quad & \langle \mathrm{id}_y \parallel k \cdot \mu z_2.\langle \mathrm{id}_x \parallel z_2 \cdot \star\rangle\rangle \\
\triangleright^+ \quad & \langle \mathrm{id}_x \parallel k \cdot \star\rangle \\
\triangleright^+ \quad & \langle k \parallel \star\rangle
\end{aligned}
$$

The significant reductions occur between the last three lines, where the functions id_y and id_x are called *in that order*. This indeed matches the order of reductions of CBV λ-calculus: the call to id_y is evaluated before the call to id_x. On the other hand, if we use the CBN semantics (noted $\triangleright^-$), evaluation goes as:

$$
\begin{aligned}
& \langle \mu a_3.\langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot a_3\rangle\rangle\rangle \parallel \mu z_2.\langle \mathrm{id}_x \parallel z_2 \cdot \star\rangle\rangle \\
\triangleright^- \quad & \langle \mathrm{id}_x \parallel \mu a_3.\langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot a_3\rangle\rangle\rangle \cdot \star\rangle \\
\triangleright^- \quad & \langle \mu a_3.\langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot a_3\rangle\rangle\rangle \parallel \star\rangle \\
\triangleright^- \quad & \langle \mathrm{id}_y \parallel \mu z_3.\langle k \parallel \mu z_4.\langle z_3 \parallel z_4 \cdot \star\rangle\rangle\rangle \\
\triangleright^- \quad & \langle k \parallel \mu z_4.\langle \mathrm{id}_y \parallel z_4 \cdot \star\rangle\rangle \\
\triangleright^- \quad & \langle \mathrm{id}_y \parallel k \cdot \star\rangle \\
\triangleright^- \quad & \langle k \parallel \star\rangle
\end{aligned}
$$

The CBN semantics has the call to id_x occurring at the second reduction (between lines 3 and 4) and the call to id_y at the last one, giving the reverse order of CBV semantics. Those two reduction sequences, exemplify the change in resolution of the two-binder conflict, which allows the machine to *selectively* assume CBV and CBN semantics. In the next section, those two semantics will be straightforwardly joined together to produce an abstract machine whose head reduction *simultaneously* embeds CBV, CBN, allowing for arbitrary compositions of the two. This will be our suitable model to recreate resource analysis in a reusable setting.

3.7 Closing remarks

Abstract machines formalize the small-step semantics of λ-terms by turning all reductions into head-reduction of the machine state. This ends up having a deep relation to resource manipulations. We've seen when describing AARA that, at a given point in a program, typing judgments endow types with potential annotation. Those annotations occur only at the root of type expressions. On the other end, CBV and CBN operational semantics require performing reduction under contexts. We can describe the problem this causes intuitively: since the type of sub-expressions appear within the type of larger expressions, reduction under context has an effect-at-a-distance on the type of the larger, surrounding expression. When using a resource-aware type system, the consistent treatment of resources therefore requires ensuring that as a program reduces, this effect on types is taken into account.

Using an abstract machine, all reduction in both CBV and CBN semantics are head reduction. Since an AARA-style type system keeps resources at the root of types, and all reductions occur on the root command, the entire treatment of resources during analysis occurs at the root of the programs being analysed. Furthermore, focusing and inverting allows us to only consider programs of a certain form, in which producing and consuming values/stacks happens as close as possible to their production without changing semantics. Thanks to the strong logical background of those methods, we will be able to embed resource information into the types of programs in natural wau, following the inversion/focusing strategy.

This will allow us to construct a consistent-by-definition resource footprint analysis from first principles, while dispensing with the most laborious proof-work. To begin this work, let us introduce the final form of our abstract machine, *the system-**L***. We shall as well introduce a type system for it that encodes operational semantics, which will turn out to match exactly with polarized intuitionistic linear logic, the fully-fledged form of the logical system we used to define the machines of this section.

Chapter 4

The System-**L** machine and Call-by-Push-Value semantics

The target of our resource analysis is a fusion of the CBV and CBN abstract machine presented in the last chapter. This machine is known in the literature under many names, and we choose to call it the System-**L**, or just **L** machine. It was initially introduced by Herbelin and Curien in [18] as a mean to elucidate which programming languages might correspond to classical logic under the Curry-Howard correspondence, in the same way proofs in minimal intuitionistic logic correspond to λ-calculus. It was made compatible with the specification of evaluation order in [19], and proposed as an intermediate representation by Downen[23]. It was the subject of a recent *Functional Pearl* as part of the *ICFP'24* conference[15]. The version of **L** we use in this manuscript is taken from [17], extended with algebraic datatype definitions and guarded fixpoints.

We now introduce this machine, its encoding of data structures, recursion, and explicit scope management. Its linear type system closely related to linear logic: taking a typing derivation of a program and erasing terms produces exactly sequent-style derivations *focused intuitionistic linear logic*, and vice versa.

4.1 Principle and main syntax

The **L** machine is, as the machines of the last chapter, made of commands c on which small-step reduction $\triangleright$ is implemented, each reduction having either CBV or CBN semantics. Commands

are made of a term t, some terms being values V, interacting with an environment e, some of them being stacks S. Those five syntactic categories are collectively called *expressions*. An η-reduction relation $\triangleright^t$ is defined on terms t as the closure under context of basic η-reductions. The same goes for environments e, on which we define the η-reduction relation $\triangleright^e$. We also define a cost-aware small-step operational semantics written $\triangleright_k$ for a cost k. We write $\triangleright$ for $\triangleright_0$ for simplicity.

All the syntax presented in the CBV (resp. CBN) machine is part of the **L** machine and we briefly re-introduce them in this new formal context. We do so by fragments, starting with control flow, introducing then data structures and computations, shifts between CBV and CBN semantics, scope management, and finally recursive computations.

Overview Commands, terms, types etc. from the CBV machines semantics are *positive*, and their CBN counterparts *negative*. Each expression is unambiguously annotated with its *polarity*, notated $+$ for CBV or $-$ for CBN. As previously we omit the positive/negative polarity annotations when unambiguous as to lighten the notation.

Just as is the CBV machine, positive commands $c^+ = \langle t^+ \parallel S^+ \rangle$ are made of a term t^+ interacting with a stack S^+. Note that some positive terms are *values* V^+, but all positive environments are stack. Dually, negative commands $c^- = \langle V^- \parallel e^- \rangle$ from the CBN machine are always formed of a negative value V^- interacting with an environment e^-, which may be a stack S^-.

Value variables are written x, y, z, and stack variables a, b, c. Both can be positive or negative. Term may have zero, one, or more free value variables, but no free stack variables. Commands/environments may have zero, one, or more free value variables, and always one free stack variables. We call this free stack variables the *final continuation* of the command/environment. Execution proceeds by reduction with capture-avoiding substitution in commands, as previously. Those substitutions are of values for value-variables and stacks for stack-variables. Note that they are *linear*: each free variable in a substitution only has one occurrence.

There are two *neutral* commands that implement scope management and costs. Namely, the command $\langle \sigma; c \rangle$ applies the variable-to-variable substitution σ to c. This substitution need not be linear, and is the only structure of the **L** machine allowing weakening (removing an unused variable) and contraction (duplicating a variable). The other neutral command is $\langle \$k; c \rangle$, for

pol.		positive/CBV					negative/CBN					neutral
c	$::=$	$\langle t^+ \parallel S^+ \rangle^+$					$\langle V^- \parallel e^- \rangle^-$					$\langle \$k; c \rangle \mid \langle \sigma; c \rangle$
$t^\pm$	$::=$	V^+	$\mid$	$\mu a^+.c$			V^-					
$e^\pm$	$::=$	S^+					S^-	$\mid$	$\mu x^-.c$			
$V^\pm$	$::=$	x^+			$\mid$	$(\cdots)^+$	x^-	$\mid$	$\mu a^-.c$	$\mid$	$(\cdots)^-$	
$S^\pm$	$::=$	a^+	$\mid$	$\mu x^+.c$	$\mid$	$(\cdots)^+$	a^-	$\mid$			$(\cdots)^-$	

$$\langle \mu a^\pm.c \parallel S^\pm \rangle^\pm \triangleright c[S/a] \qquad \textbf{(bind-}S\textbf{)}$$
$$\langle V^\pm \parallel \mu x^\pm.c \rangle^\pm \triangleright c[V/x] \qquad \textbf{(bind-}V\textbf{)}$$
$$\langle \$k; c \rangle \triangleright_k c \qquad \textbf{(cost)}$$
$$\langle \sigma; c \rangle \triangleright c\sigma \qquad \textbf{(share)}$$

$$\mu x^\pm.\langle x^\pm \parallel e^\pm \rangle^\pm \triangleright^e e \qquad \textbf{(eta-bind-}V\textbf{)}$$
$$\mu a^\pm.\langle t^\pm \parallel a^\pm \rangle^\pm \triangleright^t t \qquad \textbf{(eta-bind-}S\textbf{)}$$

$k \in \mathbb{Z}$, is a tick which induces a cost k and reduces to c (see section 2.2). The cost metric for System-**L** is the one induced by those ticks.

Control flow within a program is determined by the positive or negative polarity of its top-level command, term and environment (which are guaranteed to all match). Positive (resp. negative) polarity has CBV (resp. CBN) semantics: expressions with positive polarity are exactly those of the CBV machine introduced last chapter, and likewise, the negative constructs are exactly those of the CBN machine. The neutral commands only have a single direct sub-command, and therefore have non-ambiguous reduction without the need to be polarized. The binders still have the same reduction rules: the fact that $\mu a^+.c$ is a full term and not a value guarantees that the positive fragment has CBV semantics, and, likewise, $\mu x^-.c$ being a full environment gives the negative fragment CBN semantics.

Control structures In the positive case, the **L** machine behaves as the CBV machine: during reduction of the command $\langle \mu a^+.c \parallel \mu x^+.c' \rangle$, the term $\mu a^+.c$ dominates the stack $\mu x^+.c'$, allowing the former to capture control by binding the latter to its continuation a. In the negative case, on the other hand, goes along CBN semantics: when reducing $\langle \mu a^-. \parallel \mu x^-.c' \rangle$,

the negative environment $\mu x^-.c$ interact with the negative value $\mu a^-.c$ by capturing it as x^-, without the value being reduced.

This forms the core of the machine, which we call the *identity block*. All additions to the machine are defined by adding a new value/stack pair and a reduction rule specifying how they interact. The positive/negative polarity of this new pair determines whether an extension has CBV or CBN semantics.

4.2 Types and constraints

Type system we put on System-**L** is a sequent-style, linear presentation of Levy's simply-typed call-by-push-value λ-calculus[52], enriched with a fragment of first-order logic (FOL) at type-level. Our resource- and size-aware type system will then be defined as an enrichment of this simple type system. We split our presentation accordingly. In this chapter, we define the language of types with its FOL extension, but present only the simply typed machine. This will allow us to focus our attention to operational semantics. Then, in chapter 6 we shall present the full use of the first-order fragment, and its application to resource analysis. This allows us to bring attention to the fact that this FOL extension acts as a formalism for static analysis for a general purpose calculus.

Sorting and type language The type system separates type-level syntax in two layers: *types* and *parameters*. We use the word *type* in the traditional sense, to mean a static annotation on a expression. Types variables $A, B, C, \dots$ range over types. Types then have a *sort*: positive (resp. negative) expressions have a type of sort **pos** (resp. **neg**). The sorts **pos** and **neg** are called base sorts, and variables $\mathbf{b_1}, \mathbf{b_2}, \dots$ range over them. We annotate sorts for emphasis, writing A^+ for a type A with positive sort, A^- for a negative sort, and $A^\pm$ or A^ϵ when we wish to emphasize both are possible.

The second layer implements resource analysis using first-order logic. We call first-order terms *parameters*. Those have a *parameter sort*, such as "natural integers", which we formally denote $\mathbf{s}_1, \mathbf{s}_2, \dots$. Parameters expressions are written minuscule letter $\tau_1, \tau_3, \dots$, and parameters variables $\alpha, \beta, \gamma, \dots$ or I, J, N, M. Finally, parameter operations are written $\varphi, \psi, \dots$ and have sorts $(\mathbf{s}_1, \dots, \mathbf{s}_n) \to \mathbf{s}$.

In general, types will depend on parameters: for example, integers with statically determined upper and lower bounds will depend on two parameters. Such types are therefore are type-level functions of sort $(\mathbf{s}_1, \ldots, \mathbf{s}_n) \to \mathbf{b}$, which we call *monotypes*. We overload type variables $A, B, C, \ldots$ to also range over for monotypes and monotype variables. We also identify base sorts $\mathbf{b}$ with monotype sorts with no arguments $() \to \mathbf{b}$.

Finally, programs may define new types as type synonyms or algebraic datatype (ADT). Those definitions introduce *type constructors*. For example, linked lists, defined the usual way for functional programming, are type constructors. Type constructors are functions from monotypes to a monotype, with sort $(\mathbf{m_1}, \ldots, \mathbf{m_n}) \to \mathbf{m}$. We use variables $\mathbf{k}_1, \mathbf{k}_2, \ldots$ to denote the sorts of type constructors, and variables $K, L, \ldots$ to range over type constructors.

This yields the following grammar of sorts, types, parameters, and monotypes. Parameters are either variables or an operation applied to parameters. Monotypes are a variable, the abstraction or application of a parameter to a monotype, or a type constructor applied to a monotype. Finally, type constructors are either the hard-coded constructors ! and **fix**, which we'll introduce later, or a user-defined type constructor K. Note that while monotypes can be partially applied to yield more monotypes, type constructors must always be fully applied.

$$
\begin{array}{ll}
\mathbf{s} = \mathbf{s}_1, \cdots, \mathbf{s}_n & \textit{(parameter sorts)} \\
\mathbf{b} ::= \mathbf{pos} \mid \mathbf{neg} & \textit{(base type sorts)} \\
\mathbf{m} ::= \mathbf{b} \mid \mathbf{s} \to \mathbf{m} & \textit{(monotypes sorts)} \\
\mathbf{k} ::= (\mathbf{m}_1, \cdots, \mathbf{m}_n) \to \mathbf{m} & \textit{(type constructors sorts)} \\
\tau ::= \alpha \mid \varphi(\vec{\tau}) & \textit{(parameters)} \\
A ::= \lambda(\alpha : \mathbf{s}).A \mid A(\tau) \mid K(\vec{A}) & \textit{(types)} \\
K ::= ! \mid \mathbf{fix} \mid \ldots & \textit{(type constructors)}
\end{array}
$$

We can now express the sorting rules . Sorting occurs in a context $\Theta = \Theta_{\mathbf{s}} \cup \Theta_{\mathbf{b}} \cup \Theta_{\mathbf{m}} \cup \Theta_{\mathbf{k}}$ associating a sort to each type-level to variable (type constructors, monotypes, base types, and parameters). The rules for application and abstractions are obvious, and we provide some others below:

$$
\frac{(A : \mathbf{b}) \in \Theta_{\mathbf{b}}}{\Theta \vdash A : \mathbf{b}}\ \textbf{(sort-var)} \qquad \text{and likewise for other sorts}
$$

$$\frac{\overrightarrow{\Theta \vdash A_i : \mathbf{m_i}} \qquad K : (\mathbf{m_1}, \cdots, \mathbf{m_n}) \to \mathbf{m}}{\Theta \vdash K(A_1, \cdots, A_n) : \mathbf{m}}\ \textbf{(sort-cons)}$$

$$\frac{}{\Theta \vdash\ ! : \mathbf{neg} \to \mathbf{pos}} \qquad \frac{}{\Theta \vdash \mathbf{fix} : \mathbf{neg} \to \mathbf{pos}}$$

First-order constraints Each program is defined in the context of a first-order *signature*, which defines a set $\mathcal{S}$ of *parameter sorts*, a set $\mathcal{O}$ of *operators*, and a set $\mathcal{R}$ of *relations*. Furthermore, each n-ary operator φ in a signature has a sort $\mathrm{ar}(\varphi) = (\mathbf{s}_1, \ldots, \mathbf{s}_n) \to \mathbf{s}$, and each n-ary relation R a sort $\mathrm{ar}(R) = (\mathbf{s}_1, \ldots, \mathbf{s}_n)$. Signatures need not include equality, as which is a primitive.

Assuming a first-order signature is given, we define first-order *constraints*, which belong to a fragment of multi-sorted first-order logic. Conjunctions of relations and equalities makes up *simple constraints* $E, E', \ldots$. On top of those simple constraints, full *constraints* $C, C', \ldots$ are either *true* ($\top$), *false* ($\bot$), a conjunction or assumption of a simple constraint ($E \wedge C$ and $E \Rightarrow C$), or a quantification ($\exists\alpha.C$ and $\forall\alpha.C$). We use the standard capture-avoiding substitution on first-order syntax, written $\tau'[\tau/\alpha]$, $E[\tau/\alpha]$, and $C[\tau/\alpha]$. The well-sortedness relations $\Theta \vdash E$ and $\Theta \vdash C$ are assumed to be well-understood. This is summarized below.

First-order signatures

Sorts:	$\mathbf{s}, \mathbf{s}', \mathbf{s}'', \cdots \in \mathcal{S}$
Operators:	$\varphi, \psi, \chi, \cdots \in \mathcal{O}$
Relations:	$R, P, Q, \cdots \in \mathcal{R}$
Operator arity:	$\mathrm{ar}(\varphi) = (\mathbf{s}_1, \ldots, \mathbf{s}_n) \to \mathbf{s}'$
Relation arity:	$\mathrm{ar}(R) = (\mathbf{s}_1, \ldots, \mathbf{s}_n)$

First-order constraints

Simple constraints:	$E ::= \top \mid R(\vec{\tau}) \wedge E \mid \tau = \tau' \wedge E$
Constraints:	$C ::= \top \mid \bot \mid E \wedge C \mid E \Rightarrow C \mid \forall\overrightarrow{\alpha : \mathbf{s}}.\, C \mid \exists\overrightarrow{\alpha : \mathbf{s}}.\, C$

4.3 Defining types

Type definitions, extended with parameters and constraints, form the foundation of the reasoning power within System-**L**. They come in four flavors. First, **type** defines a new type synonym, i.e. a type constructor interchangeable with a type it abbreviates. Then, **newtype** defines a type synonym with much be explicitly boxed and unboxed at compile-time, but have

the same runtime representation as the type they are defined from, *à la* Haskell. Finally, **data** and **comput** define algebraic types. We give here the sorting rules that characterize valid definitions, but we shall not use those until later in the chapter. The syntax not part of the simple type system are highlighted in blue. They can be safely ignored in this chapter.

We assume programs contain a "prelude" that simultaneously defines many mutually recursive type constructors. As to allow the mutually recursive definition of types constructors, the sort of each type constructor is explicitly stated in its definition. The first step of checking type definitions is therefore to collect all type constructors K and their respective sorts $\mathbf{k}$ to create a first scope $\Theta_{\mathbf{k}}$ of all type constructors. Each definition is then checked within this scope. The second preliminary step is thereby to check that all expression-level constructors declared within type definitions are distinct. With this in mind, let us present and describe the well-sortedness requirement of type definitions.

type definitions The definitions of new **type**-s introduce a new type constructor $K(\dots)$ from a type expression B. At runtime, both $K(\dots)$ and B have the same representations and are interchangeable. The syntax for **type** definitions is shown below. The defined type constructor K has monotype arguments $\overrightarrow{A_i : \mathbf{m_i}}$ and parameter arguments $\overrightarrow{\alpha_j : \mathbf{s_j}}$, and is declared to have a base sort $\mathbf{b}$ when fully applied. This means K can be seen equivalently as a base type of sort $\mathbf{b}$ which depends on $\overrightarrow{A_i}$ and $\overrightarrow{\alpha_i}$, or as a mapping from monotypes $\overrightarrow{A_i}$ to a monotype of sort $(\mathbf{s}_1, \dots, \mathbf{s}_m) \to \mathbf{b}$.

$$\texttt{type}\ \ K(A_1 : \mathbf{m_1}\,,\ \dots,\ A_n : \mathbf{m_n})\ \ (\alpha_1 : \mathbf{s_1},\ \dots,\ \alpha_m : \mathbf{s_m})\ \ :\ \ \mathbf{b} = B$$

A **type** definition as shown above contains a single type expression B as its body. This expression can contain type variables A_i, parameter variables α_i, and any type constructor declared in the prelude. Formally, the following rule formalizes the validity of **type** definitions:

$$\frac{\Theta_{\mathbf{k}}, \overrightarrow{A_i : \mathbf{m}_i}, \overrightarrow{\alpha_i : \mathbf{s}_i} \vdash B : \mathbf{b}}{\textbf{type}\ K(\overrightarrow{A_i : \mathbf{m}_i})(\overrightarrow{\alpha_i : \mathbf{s}_i}) : \mathbf{b} = B\ \textit{is a valid def.}}$$

newtype definition While a **type** acts as a strict synonym for its definition body **newtype** creates a distinct type from its definition body at compile-time, but with the same runtime representation. Those are inspired from the feature of the same name in Haskell, and correspond to a variation on the one-constructor datatypes found in OCaml. Just as for **type** definitions,

the type constructor introduced in a **newtype** definition can either be seen as a base type with arguments, or a mapping between monotypes. Their syntax is shown below:

newtype $K(A_1 : \mathbf{m_1}\,,\ \ldots,\ A_n : \mathbf{m_n})\ (\alpha_1 : \mathbf{s_1}\,,\ \ldots,\ \alpha_m : \mathbf{s_m})\ :\ \mathbf{b} = k$ **of** B
  where $\beta_1 : \mathbf{s'_1}\,,\ \ldots,\ \beta_p : \mathbf{s'_p}$
  with E

To go back and forth between a value/stack of type $K(\ldots)$ to one of type B, one must box or unbox the expression-level constructor k defined with the **newtype**. Furthermore, that expression-level constructor introduces and eliminates some parameters $\overrightarrow{\beta_j}$ defined in a **where** and a simple constraint E defined in a **with** clause. Both those clauses are optional, in which case we take $\overrightarrow{\beta_k}$ to be empty and E to be $\top$. Those two new clauses are irrelevant for now, and **newtype** can safely be understood as one-constructor datatype in this chapter.

As far as sorting is concerned, E is checked in context of the $\overrightarrow{\alpha_i}$ and $\overrightarrow{\beta_k}$, and B is checked in context of all type-level variables in the definition. The rule describing the validity of **newtype** definitions is therefore:

$$\frac{\Theta_{\mathbf{k}}, \overrightarrow{A_i : \mathbf{m}_i}, \overrightarrow{\alpha_j : \mathbf{s}_j}, \overrightarrow{\beta_k : \mathbf{s}_k} \vdash B : \mathbf{b} \qquad \overrightarrow{\alpha_j : \mathbf{s}_j}, \overrightarrow{\beta_k : \mathbf{s}_k} \vdash E}{\textbf{newtype}\ K(\ldots)(\ldots) = \ldots \ \textit{is a valid def.}}$$

datatype definitions Datatypes use constructors to represent algebraic data structures with CBV semantics. Those include pattern-matching pairs and sum types. Each datatype definition in the prelude exports a type constructor and a finite set of value constructors. An example definition for linked-lists is shown below on the left, and the full syntax of datatype definitions is shown on the right.

```
data List(A:pos) =
  | cons of A ⊗ List(A)
  | nil (* no arguments*)
end
```

data $K(\vec{\mathsf{A}} : \vec{\mathbf{m}})$ $(\vec{\alpha} : \vec{\mathbf{s}})$ =
  | k_1 **of** $\overrightarrow{\mathsf{B}_1}$ **where** $\overrightarrow{\beta_1} : \overrightarrow{\mathbf{s}'_1}$ **with** E_1
  ...
  | k_m **of** $\overrightarrow{\mathsf{B}_m}$ **where** $\overrightarrow{\beta_m} : \overrightarrow{\mathbf{s}'_m}$ **with** E_m
end

This syntax should be intuitive to the readers versed in functional programming with languages of the ML extended family. Note the use of the *linear* tuple operator $\otimes$. Our type system is linear, and ownership of a list value of the form $\mathtt{cons}(h, t)$ is identical to the ownership of its head h and tail t. In general, a type defined with **data** is an n-ary version of a **newtype**: it has many constructors, each with many arguments and its own **where** and **with** clause. When

sorting, the same rule as for **newtype** apply, with the restrictions that $K(\dots)$ always has sort **pos**, and that all arguments $\beta_{i,j}$ are positive as well. This gives the sorting rule below.

In the following, we omit the indexing on variables A, α, $\mathbf{m}$ and $\mathbf{s}$, and the sort of type variables, which are identical to those for the **newtype** definition. All premises that depend on index i should be iterated over all constructors, and those depend on (i, j) should be twice-iterated: once over constructors, and once over the arguments of each constructor.

$$\frac{\Theta_{\mathbf{k}}, \vec{A}, \vec{\alpha}, \vec{\beta_i} \vdash B_{i,j} : \mathbf{pos} \qquad \vec{\alpha}, \vec{\beta_i} \vdash E_i}{\textbf{data}\ K(\dots)(\dots) = \dots \ \textit{is a valid def.}}$$

computation type definitions Computation types are the lazy counterpart of datatypes. Namely, whereas datatypes are defined by the constructors that they support, computation types are defined by the accessors they support, that is, by what queries they can answer. As opposed to datatype values, which are evaluated as early as possible (CBV, sort **pos**), the result of accessing a computation type is computed as late as possible (CBN, sort **neg**). An example definition of a stream type is given below.

```
comput Stream(A:pos, B:pos) =
  | Charge of 𝟙 ⊸ Stream(A, A ⊗ B)
  | Get of 𝟙 ⊸ ⇑B ⊗ ⇓Stream(A, 𝟙)
end
```

The Stream computation type has two accessors: charge computes a new value of type A and adds it to the already-generated values. Successive calls to charge loads-up larger tuples of values, all of type A. Once a required number of values have been generated, they are simultaneously returned by accessing get. This returns a tuple of the loaded values and another stream, ready produce the next values. The $\mathbb{1} \multimap$ denotes the fact that those accessors do not take any arguments. For now, we may ignore the two arrow $\Uparrow$ and $\Downarrow$, which handle the management of the lazy stream and eager values it generates. Let us now move on to the full syntax of definitions.

```
comput K(A⃗:m⃗) (α⃗:s⃗) =
  | k_1 of B_{1,1} ⊗ ⋯ ⊗ B_{1,k_1} ⊸ D_1  where β⃗_1:s⃗'_1 with E_1
  ...
  | k_m of B_{m,1} ⊗ ⋯ ⊗ B_{m,k_m} ⊸ D_m  where β⃗_m:s⃗'_m with E_m
end
```

Just as datatypes, computation types have monotype arguments $\vec{A}$ and parameter arguments $\vec{\alpha}$. Each constructor $k_i$they define also introduces parameters $\vec{\beta_i}$ subject to a simple constraint E_i. As opposed to datatypes, those constructors represent accessors. They are called with arguments $\vec{B_i}$ with sort **pos** and have return type D_i of sort **neg**. The sorting rules are similar to those of datatypes, with changes made to accommodate the return type and the overall negative sort:

$$\frac{\Theta_{\mathbf{k}}, \vec{A}, \vec{\alpha}, \vec{\beta_i} \vdash B_{i,j} : \mathbf{pos} \qquad \Theta_{\mathbf{k}}, \vec{A}, \vec{\alpha}, \vec{\beta_i} \vdash D_i : \mathbf{neg} \qquad \vec{\alpha}, \vec{\beta_i} \vdash E_i}{\mathbf{data}\ K(\dots)(\dots) = \dots \ \textit{is a valid def.}}$$

Typing constructors in programs This concludes our presentation of type definitions. As to be able to use them in programs in a well-typed manner, we should provide some link between the type given to constructors in definitions and their use in program. As such, we introduce two judgments, which should be familiar to readers acquainted with the implementation of ML-style programming languages. The first one is *constructor exhaustivity*. We say that a list of constructors $k_1, \dots, k_n$ is exhaustive for a type constructor K when that list contains exactly the constructors defined as part of K, without duplicates. As such, a pattern-matching expression with cases on constructors $k_1, \dots, k_n$, then, it is total and non-ambiguous for expressions of type $K(\dots)$ iff the constructors $k_1, \dots, k_n$ exhaust K. We write this judgment $k_1, \dots, k_n \twoheadrightarrow K$ and define it as

$$\frac{\textit{Each } k \textit{ defined for } K \textit{ is some } k_i. \qquad \textit{All } k_i \textit{ are distincts.}}{k_1, \cdots, k_n \twoheadrightarrow K}$$

The second judgment relates the type of a constructor in its definition to its type in programs. Indeed, constructors in programs don't *literally* use the type and parameter variables used in their definition, but instead are used with consistent instantiations of those variables. In general, a such an instantiation for a datatype constructor k^+ (resp. computation type constructor k^-) involves the monotypes arguments $\vec{A}$ and parameters arguments $\vec{\alpha}$ of its type constructor K, the parameters $\vec{\beta}$ and simple constraint E it introduces and its arguments $\vec{B}$ (resp. its arguments $\vec{B}$, and return type D). If all those are correctly instantiated, we write:

$$\vdash k^+ : \exists \vec{\beta}.E \wedge \vec{B} \to K(\vec{A})(\vec{\alpha})$$

$$\vdash k^- : \exists \vec{\beta}.E \wedge \vec{B} \multimap D \to K(\vec{A})(\vec{\alpha})$$

Not all constructors involves all those variables: in the simple type system, $\vec{\alpha}$ and $\vec{\beta}$ are empty and E is trivial; in datatypes, D is absent; and, in both computations and datatypes, the list

of arguments $\vec{B}$ may be empty. In those cases, we either replace the corresponding elements with the symbol $\emptyset$, or remove them from the judgment altogether, leading to judgments such as:

$$\vdash \mathsf{nil} : () \to \mathsf{List}(A)$$

$$\vdash \mathsf{cons} : (\emptyset, \emptyset, (A, \mathsf{List}(A))) \to \mathsf{List}(A)$$

$$\vdash \mathsf{charge} : \mathsf{Stream}(A, A \otimes B) \to \mathsf{Stream}(A, B)$$

The rules defining the instantiation judgment encode that an instantiation is valid for a given constructor when its contents are litterally those of the constructor's definition, and that instantiations remain valid after applying a substitution σ sending $\vec{\alpha}$ to paramters and $\vec{A}$ to monotypes (note that the bound parameter β cannot be instantiated). This gives the following two rules for datatypes, and similar rules for computation types.

$$\frac{\vdash k^+ : \exists \vec{\beta}.E \wedge \vec{B} \to K(\vec{A})(\vec{\alpha})}{\vdash k : \exists \overrightarrow{\beta}.E\sigma \wedge \overrightarrow{B\sigma} \to K(\overrightarrow{A\sigma})(\overrightarrow{\alpha\sigma})}$$

$$\frac{k \textit{ is literally defined as such.}}{\vdash k : \exists \vec{\beta}.E \wedge \vec{B} \to K(\vec{A})(\vec{\alpha})}$$

This closes our presentation of type definitions. The rest of the chapter is dedicated to the expression-level syntax and semantics of the machine, and their type system.

4.4 The simple type system

In this section, we introduce the simple type system of System-**L**. For now, only the judgments and rules for the simple type system are described, as we reverse the treatment of parameters and constraints for the next chapter. We begin with specifying the judgments used for the different syntactic classes of expressions.

Judgments Terms, environments and commands are typed in context of a parameter scope $\Theta = \Theta_{\mathbf{s}}$, a value scope $\Gamma = \overrightarrow{x : A}$ which associate a base type to each value variable in scope, and a continuation with a base type $\Delta = (a : A)$. Moreover, each typing judgment is done relatively to a constraint C which the parameters in Θ must satisfy. This means that typing judgments are all of the form $(\Theta \models C) \triangleright (\Gamma \vdash \Delta)$, which can be thought of as "for all

parameters in Θ which satisfy C, a program with inputs Γ provides an output on Δ". This can be seen as a two-level system, where first-order logic sits on top of linear, intuitionistic sequent calculus. When this upper level is trivial – that is when $\Theta = \emptyset$ and $C = \top$ – we omit it and write $\Gamma \vdash \Delta$. We call this a *simple judgment*. The typing rules of simple judgment follow those of [17], and form a term assignment system for *focused intuitionistic linear logic*. Without further ado, the five typing judgment in the system go as follows:

Sort	Full judgement	Simple judgement
Commands	$c :: (\Theta \models C) \triangleright (\Gamma \vdash \Delta)$	$c : (\Gamma \vdash \Delta)$
Terms	$(\Theta \models C) \triangleright \Gamma \vdash t : A$	$\Gamma \vdash t : A$
Values	$(\Theta \models C) \triangleright \Gamma \vdash V : A$	$\Gamma \vdash V : A$
Environments	$(\Theta \models C) \triangleright \Gamma \mid e : A \vdash \Delta$	$\Gamma \mid e : A \vdash \Delta$
Stacks	$(\Theta \models C) \triangleright \Gamma \mid S : A \vdash \Delta$	$\Gamma \mid S : A \vdash \Delta$

Translation from simple types Most rules in the type system do not manipulate the first-order logic fragment $\Theta \models C$. This means they can be expressed in the simple type system without loss of generality, which we shall take advantage of. To this end, we formalize how rules in the simple type system extend to rules in the fully-featured one.

Consider a generic typing rule with premises $(\Gamma_i \vdash \Delta_i)$ and conclusion $(\Gamma \vdash \Delta)$. In the full type system, each sequent of the premises becomes enriched with a FOL fragment, and becomes $(\Theta_i \models C_i) \triangleright (\Gamma_i \vdash \Delta_i)$. In this situation, the parameter scopes of each premise are concatenated, and their corresponding constraints are combined as a conjunction. This gives a FOL fragment $\cup_i \Theta_i \models \wedge_i C_i$, which then enriches the conclusion, giving a judgment $(\cup_i \Theta_i \models \wedge_i C_i) \triangleright (\Gamma \vdash \Delta)$. Visually, this generic transformation rule is:

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta} \longmapsto \frac{(\Theta_1 \models C_1) \triangleright (\Gamma_1 \vdash \Delta_1) \quad \cdots \quad (\Theta_n \models C_n) \triangleright (\Gamma_n \vdash \Delta_n)}{(\cup_i \Theta_i \models \wedge_i C_i) \triangleright (\Gamma \vdash \Delta)}$$

This automatic lifting of rules from the simple type system to the full one will allow us to omit the management of parameters and constraints in this chapter, simplifying our presentation of System-**L**.

First typing rules We now proceed with the typing rules for the syntax we have introduced so far. The judgments for the terms, environments, values, stacks and commands defined so

far form the *identity group*. Rules for variables are straightforward. Note that the scopes Γ and Δ must be empty of restricted only to the variable being typed, as the type system is linear.

$$\frac{}{x : A \vdash x : A}\ \textbf{(id-var-R)} \qquad \frac{}{\emptyset \mid a : A \vdash a : A}\ \textbf{(id-var-L)}$$

Commands are typed using the *cut* rule. Polarization is exhibited as the difference between terms/environments and values/stacks. The two polarities guarantee that terms and environments never interact directly, which gives an unambiguous direction to cut-elimination at type-level, matching our small steps semantics.

$$\frac{\Gamma \vdash t : A^{\pm} \qquad \Gamma' \mid e : A^{\pm} \vdash \Delta}{\langle e \parallel t \rangle : (\Gamma, \Gamma' \vdash \Delta)}\ \textbf{(cut)}$$

In System-**L**, terms (resp. environments) are either a strong binder or a value (resp. stacks), the latter also including a weak binder differing from the strong one only in polarity. The binders have no effects at type-level, merely allowing passing between commands and terms or environments.

$$\frac{c : (\Gamma \vdash a^{+} : A^{+})}{\Gamma \vdash \mu a^{+}.c : A^{+}}\ \textbf{(id-str-R)} \qquad \frac{c : (\Gamma, x : A^{-} \vdash \Delta)}{\Gamma \mid \mu x^{-}.c \vdash \Delta}\ \textbf{(id-str-L)}$$

$$\frac{c : (\Gamma \vdash a^{-} : A^{-})}{\Gamma \vdash \mu a^{-}.c : A^{-}}\ \textbf{(id-weak-R)} \qquad \frac{c : (\Gamma, x : A^{+} \vdash \Delta)}{\Gamma \mid \mu x^{+}.c \vdash \Delta}\ \textbf{(id-weak-L)}$$

4.5 Using datatypes

We can now move on to the use of datatypes in programs and their typing rules. They behave as algebraic datatypes usually do in a linear functional language. We begin with introducing standard datatypes definable in our system.

Standard datatypes Using constructors, common datatypes in linear functional programming can be defined, namely: pattern-matching pairs and the unit type, and sum types with their unit the empty type. The pattern-machine pair (V, V') of type $A^{+} \otimes B^{+}$ is defined by a single, two-argument constructor, written infix. We shall also use n-ary tuples $(V_1, \cdots, V_n)$ of type $A_1 \otimes \cdots \otimes A_n$, also written $(\vec{V_i}) : \overrightarrow{A_i^{\otimes}}$ when notation requires an economy of space. The unit datatype $\mathbb{1}$ is a just a tuple $()$ of zero values. Tuples are defined as:

```
data Tuple(A₁ : pos, ⋯, Aₙ : pos) =
  tuple of A₁ ⊗ ⋯ ⊗ Aₙ
end
```

The sum datatype has two constructors, $\iota_{1/2}$ and $\iota_{2/2}$, each taking one argument, with type $A \oplus B$. This corresponds to the `Either` datatype in Haskell. It also exists in n-ary variants, with constructors $\iota_{i/n}$ for $1 \leq i \leq n$ and type $A_1 \oplus \cdots \oplus A_n$ or $\overrightarrow{A_i^{\oplus}}$. The zero datatype is a sum type with zero constructors (and therefore empty), with type written $\mathbb{0}$.

```
data Sum(A₁ : pos, ⋯, Aₙ : pos) =
  ι₁/ₙ of A₁ | ⋯ | ιₙ/ₙ of Aₙ
end
```

Usage of data constructors Formally, the syntax for constructors and pattern matching is given below. In detail, the value $k^{+}_{\vec{\tau}}(\overrightarrow{V^{+}})$ represents the application of the constructor k^{+} to the (positive) argument values $\overrightarrow{V^{+}}$ and type-level parameters $\vec{\tau}$. We index constructors with k^{i} instead of k_i. Type-level parameters can be ignored for now. Only fully applied constructors are authorized (i.e. no constructor currying is allowed). The value $\mathbb{0}$ and stack $\mu\mathbb{0}_{\Gamma,\Delta}$ acts as an absurd value and absurd stack, and are invalid in the simple type system. Those will become relevant when we introduce size-aware types. For example, we will be able to derive a value $\mathbb{0}_{\Gamma}$ from an inconsistent constraint such as $1 = 2$, all within a scope Γ of variables. The scopes within those absurd expression merely exists to preserve linearity.

$$V^{+} ::= k_{\vec{\tau}}(V_1^{+}, \cdots, V_n^{+}) \mid \mathbb{0}_{\Gamma}$$
$$S^{+} ::= \mu(k^{1}_{\overrightarrow{\beta_1}}(\overrightarrow{x_1^{+}}).c_1, \cdots, k^{n}_{\overrightarrow{\beta_n}}(\overrightarrow{x_n^{+}}).c_n) \mid \mu\mathbb{0}_{\Gamma,\Delta}$$

The typing rules for constructors is shown below: a constructor application is valid if each argument's type follows the specification of the constructor under some instantiation.

$$\frac{\overrightarrow{\Gamma_i \vdash V_i : U_i} \qquad \vdash k : \overrightarrow{U_i} \to K(\overrightarrow{T_j})}{\overrightarrow{\Gamma_i} \vdash k(\overrightarrow{V_i}) : K(\overrightarrow{T_j})}\ \textbf{(data-R)}$$

(no rule for $\mathbb{0}_{\Gamma}$)

Usage of pattern-matching The stack $\mu(\overrightarrow{k_i(\vec{x_i}).c_i})$ represents case analysis on constructors: each clause $k_i(\vec{x_i}).c_i$ represents one case, that matches on k_i, binds arguments to $\vec{x_i}$, and

passes control to the command c_i where the newly-bound variables are in scope. When the stack has only one clause, we may also write $\mu k(\vec{x}).c$, eliding the parenthesis around the singleton set of clauses. The pattern-matching must be exhaustive, and each clause must match the specification of its constructor. As to simplify the presentation, we introduce a purely administrative judement which states that a clause $k(\vec{x_i}).c$ forms valid *clause* for pattern matching:

$$\Gamma \mid k(\vec{x_i}).c : K(T_j) \vdash \Delta \ \mathbf{cl}.$$

Here, K is always a type constructor and $\vec{T_j}$ are always the types its arguments. With this, the rule **(data-L)** assert that a pattern-matching stack is exhaustive, and checks that each clause is valid *under the same instantiation* of the constructor. The rule **(clause-L)** then check that a clause correctly binds each argument x_i of its constructor k with the correct instantiated type, and that its body c is valid.

$$\frac{\vec{k_i} \twoheadrightarrow K \qquad \overrightarrow{\Gamma \mid k_i(\vec{x_i}).c_i : K(\vec{T_j}) \vdash \Delta \ \mathbf{cl}.}}{\Gamma \mid \mu(\overrightarrow{k_i(\vec{x_i}).c_i}) : K(\vec{T_j}) \vdash \Delta} \ \mathbf{(data\text{-}L)}$$

$$\frac{\vdash k : (\vec{U}) \to K(\vec{T}) : \mathbf{pos} \qquad c : (\Gamma, \overrightarrow{x : U} \vdash \Delta)}{\Gamma \mid k(\vec{x}).c : K(\vec{T}) \vdash \Delta \ \mathbf{cl}.} \ \mathbf{(clause\text{-}L)}$$

The stack $\mu\mathbb{0}_{\Gamma,\Delta}$ matches the absurd datatype with no constructor. This is used to encode programs when no possible cases apply, i.e. dead code. It is annotated with a typed scope Γ and a continuation scope Δ which allow the linearity discipline to be preserved when matching on the empty type.

$$\frac{}{\Gamma \mid \mu\mathbb{0}_{\Gamma,\Delta} : \mathbb{0} \vdash \Delta} \ \mathbf{(zero\text{-}L)}$$

(no rule for $\mathbb{0}_\Gamma$)

Reduction rules All being said and done, the runtime behavior of datatypes is as expected. Both side interact by matching the constructor with the corresponding clause, which must exists due to the exhaustivity check, binds each argument and parameter to its variable, and proceeds with the body of the clause. This is rule **(data)**. There is an η-law for pattern-matching stack, which states that matching, unpacking, then repackaging a constructor value is a no-op.

The empty data-type implements the principle of explosion at runtime: when $\mathbb{0}$ interacts with its stack $\mu\mathbb{0}$, any well-type command $c_{\vec{x},\vec{y},a}$ in the same scope may follow. This is the strict equivalent to the elimination rule of this same empty type in λ-calculus. Finally, the η-rule for the empty type states that the stack for the empty stack can likewise be replaced with any well-typed positive environment in the same scope.

$$\langle k^i_{\vec{\tau}}(\vec{V}) \parallel \mu(\overrightarrow{k^j_{\vec{\beta_j}}(\vec{x_j}).c_j})\rangle^+ \triangleright c_i[\overrightarrow{V_i/x_i}, \overrightarrow{\tau/\beta_i}] \qquad \textbf{(data)}$$

$$\langle \mathbb{0}_{\vec{x}} \parallel \mu\mathbb{0}_{\vec{y},a}\rangle^+ \triangleright c, \textit{ for any } c : (\Gamma \vdash \Delta) \qquad \textbf{(zero)}$$

$$\mu(\overrightarrow{k^j_{\vec{\beta_j}}(\vec{x_j}).\langle k^j_{\vec{\beta_j}}(\vec{x_j}) \parallel e^+\rangle}) \triangleright^t e^+ \qquad \textbf{(eta-data)}$$

$$\mu\mathbb{0}_{\vec{x},a} \triangleright^e e^+, \textit{ for any } \Gamma \mid e : \mathbb{0} \vdash \Delta \qquad \textbf{(eta-zero)}$$

This concludes the term-level and runtime behavior of datatypes. We now move on the same description for computation-types. Those are symmetric to those for datatypes, but with the left and right side swapped.

4.6 Computation types

In-keeping with the theme of symmetry of System-**L**, computation types are defined using constructors, but on the right side. Namely, constructors push data on top of the stack, which creates a call site for a computation with CBN semantics. This call site is captured by the left side, which uses pattern matching to pop data off the stack.

Principle Computation types are dual to data types. Let us explain: *datatypes* are defined by a set of constructors from which values can be iteratively constructed. The stacks they interact with are completely determined by this left-side definition: they merely match and deconstruct one layer of the value. This is reversed for *computation* types: a set of constructors for the stack is defined, which allows call sites to be iteratively constructed. The value side only matches and deconstruct this stack. Computation types are then the stack-centric counterpart to data types. A computation may therefore answer many different calls. In this aspect, they mimic objects in object-oriented programming, which have many methods they that can be called on them.

Formally, a constructor k is pushed on top of a stack S^- together with positive (i.e. data) arguments and parameters to yield a new stack $K_{\vec{\tau}}(\overrightarrow{V^+}) \cdot S^-$. On the left side, the value is merely a set of pattern-matching clauses. First-order parameters are also present in the syntax, but aren't important for now. As to reduce the number parenthesis involved in clauses, we write a clause $k(\vec{x}; a).c$ as opposed to $(k(\vec{x}) \cdot a).c$, and $k(a).c$ as opposed to $(k \cdot a).c$ when k has no value arguments. There is also an empty computation $\top$ with no defined constructor, which is the negative counterpart to $\mathbb{0}$, and functions the same way. Overall, the grammar for computation types is given by:

$$V^- ::= \mu(k^1_{\vec{\beta}_1}.(\overrightarrow{x_1^+}; a_1^-).c_1, \cdots, k^n_{\vec{\beta}_n}.(\overrightarrow{x_n^+}; a_n^-).c_n) \mid \mu\top_\Gamma$$
$$S^- ::= k_{\vec{\tau}}(V_1^+, \cdots, V_n^+) \cdot S^- \mid \top_{\Gamma,\Delta}$$

Constructing calls Given a stack S^-, a constructor $k(\overrightarrow{V^+})$ can be pushed on it. This is reminiscent of *stack frames*, a construct in low-level implementation of functional languages in which the stack is split in level called *frames* containing arguments and local variables of routines. For the pushed constructor to be valid, it has to satisfy typing rule **(comput-L)**, which requires each argument to be well-typed and consistent with the definition of the constructor k, and demands the stack's type is also valid by that same definition. The constructor k is to be understood as a "tag" specifying how to dynamically dispatch the call to its computation. The constructor $\top$ for empty computation is not valid in this simple type-system.

$$\frac{\overrightarrow{\Gamma_i \vdash V_i : U_i} \qquad \Gamma' \mid S : U' \vdash \Delta \qquad \vdash k : \overrightarrow{U_i} \multimap U' \to K(\vec{T}) : \mathbf{neg}}{\overrightarrow{\Gamma_i}, \Gamma' \mid k(\vec{V_i}) \cdot S : K(\vec{T}) \vdash \Delta}\ \textbf{(comput-L)}$$

(no rule for $\top$)

Semantics Reduction proceeds for computations exactly as for the data-types, with the addition of the continuation passing (rules **(comput)** and **(eta-comput)**). Note that, when finishing a computation, that is, when jumping to its continuation, the machine remains in negative mode. This means that when a computation is over, the machine proceeds with another one by default rather than producing a return value. A computation has then the effect of iteratively consuming the calls that are required of it but never more than this, in typical CBN fashion. The empty computation $\top$ implements the principle of explosion: since

it has no stack constructors, computations of type $\top$may proceed with any command. This is formalized with the rules **(top)** and **(eta-top)**.

$$\langle \mu(\overrightarrow{k^i_{\overrightarrow{\beta_i}}(\overrightarrow{x_i}; a_i).c_i}) \parallel k^j_{\overrightarrow{\tau}}(\overrightarrow{V}) \cdot S^- \rangle^- \triangleright c_j[\overrightarrow{V/x_j}, S/a_j, \overrightarrow{\tau/\beta_j}] \qquad \textbf{(comput)}$$

$$\langle \mu\top_\Gamma \parallel \top_{\Gamma',\Delta} \rangle^- \triangleright c \qquad \textbf{(top)}$$

$$\mu(\overrightarrow{k^i_{\overrightarrow{\tau_i}}(\overrightarrow{x_i}; a_i).\langle t^- \parallel k^i_{\overrightarrow{\tau_i}}(\overrightarrow{x_i}) \cdot a_i \rangle}) \triangleright^t c \qquad \textbf{(eta-comput)}$$

(no eta-law for $\top$.) **(eta-top)**

Standard computation types Let us see some example of computations, namely functions, choices, and the never returning computation $\mu\top$.

Functions in the **L** machines are uncurried. They have a single stack constructor, "call", pushing their arguments on the stack. The name of this constructor is often elided, and we write $(V_1, \ldots, V_n) \cdot S$ for a stack where n function arguments are pushed onto of a stack S. Function literal merely match on that stack, and have syntax $\mu(x_1, \ldots, x_n; a).c$. The arity requirement means that partial application is not possible (and no currying is possible).

```
comput Func(A₁ : pos , ··· , Aₙ : pos , B : neg) =
  call of A₁ ⊗ ··· ⊗ Aₙ ⊸ B
end
```

Projections pairs are another computation type. They have two stack constructors $\pi_{1/2} \cdot S$ and $\pi_{2/2} \cdot S$, providing two distinct calls that a computation on the left side provides. Such a computation is implemented as $\mu(\pi_{1/2}(a).c, \pi_{2/2}(b).c')$. They also exist in n-ary variants with projections $\pi_{i/n}$ for $1 \leq i \leq n$. In the limit case of $n = 0$, we get the non-callable computation $\mu\top_\Gamma$ serves the same purpose as $\mu\mathbb{0}_{\Gamma,\Delta}$. It serves to encode dead continuations. Since there is only one continuation in scope, if it is dead evaluation cannot proceed. This represents global failure of the machine, as opposed to the zero type, which represent local failure (only one code path is dead).

```
comput Sum(A₁ : neg , ··· , Aₙ : neg) =
  π_{1/n} of 𝟙 ⊸ A₁ | ··· | π_{n/n} of 𝟙 ⊸ of Aₙ
end
```

As we have said, computations only ever consume the stack, without "returning" a value. Dually, data-type never consume stack constructors. To recover the full power of the CBV and CBN machines, one must allow calling computations from positive code, and return data from negative code. In other words, while we currently have a data machine (positive, CBV semantics) and a computation machine (negative, CBN semantics) side-by-side, we still have to express going back-and-forth between the two. We now tackle this fusion, which will finally endow the machine with its full-powered semantics. This will allow us to recover the power or mixed-semantics λ-calculus, together with simpler handling of resources thanks the abstract machine formalism.

4.7 Focusing with shifts and closures

As opposed to the CBV and CBN machines, in which all reductions go along a one of the two semantics, the **L** machine allows programs to deterministically mix them in a syntax-directed manner. Polarity determines oeprational semantics, and is preserved across evaluation. But mixing those semantics is required to return data from a computation or pass a computation to a high-order function. This is the goal of *shifts*, which complete the identity block.

Principle Positive and negative polarities respectively denote CBV and CBN semantics within System-**L**. When expressing CBV semantics in a continuation-passing style, we say that an expression captures its continuation, then proceeds through its evaluation, and eventually passes its value to the continuation. On the other hand, CBN in continuation-passing style imposes that when an expression interacts with a continuation, the continuation takes control and proceeds with its evaluation straight away. Boiling it down, positive terms "win control" over continuations, and negative continuations "win control" over terms. This forms the basis of the two-faced operational semantics of **L**: choosing polarity determines which sides "wins" control over the other. *Shifts* allow programs to temporarily assume another polarity as to delay or force their evaluation.

Shifts were introduced by Levy in his Ph.D. thesis[52] in the context of λ-calulus, and form the core of *Call-by-Push-Value* semantics, which subsumes CBV and CBN semantics, at the cost of a straightforward syntactic transformation of program. This system was ported by Munch-Maccagnoni et al. in[17] to the **L** machine. Shifting allow negative values/stack to

temporarily bear CBV semantics, and positive values/stacks to temporarily bear CBN ones. This is done by encapsulating them in a value/stack of the opposing polarity.

Closures and thunks In a nutshell, closures and thunks are one-arguments constructors who encapsulate a value or stack while changing its polarity. Given a computation, represented as a *negative* value V^-, the positive value $(\Downarrow V^-)^+$ is the *closure* over V. This closure can be *opened* to recover the inner computation (and return to negative mode) by deconstructing it with the stack $\mu\Downarrow x^-.c$, in which the negative variable x^- is bound to the computation, and control flow proceeds to c. The syntax, typing and reduction for closures is:

$$\begin{array}{rcl} A^+ & ::= & \Downarrow A^- \\ V^+ & ::= & \mu\Downarrow V^- \\ S^+ & ::= & \mu\Downarrow x^-.c \end{array}$$

$$\frac{\Gamma \vdash V^- : A^-}{\Gamma \vdash \Downarrow V^- : \Downarrow A^-}\ \textbf{(closure-R)} \qquad \frac{c : (\Gamma, x^- : A^- \vdash \Delta)}{\Gamma \mid \mu\Downarrow x^- : \Downarrow A^- \vdash \Delta}\ \textbf{(closure-L)}$$

$$\langle \Downarrow V \parallel \mu\Downarrow x.c\rangle \triangleright c[V/x] \qquad \textbf{(closure)}$$

$$\mu\Downarrow x.\langle \Downarrow x \parallel e^+\rangle \triangleright^e e \qquad \textbf{(eta-closure)}$$

Thunks are the formal dual of closures: they wrap a positive stack, which would otherwise consume a value, inside a negative stack whose topmost frame blocks the computation. The stack then waits for the thunk to run before consuming a value. Formally, a positive stack S^+ is wrapped inside a call to the thunk by the syntax $\Uparrow \cdot S^+$. On the other side, a thunk is a computation that matches on $\Uparrow$, binds the stack as a^+, and proceed to evaluate itself as the command c. It is written $\mu\Uparrow a^+.c$. Since a must be the only stack variable in c, it represents the final continuation of c: when triggered, the thunks runs c, which must eventually produce a positive value which interacts with a (now S). The inner command is thereby blocked from evaluating until the $\Uparrow \cdot S$ stack enables its execution, and S is blocked from consuming this result until it is produced. The syntax and rules for thunks is shown below:

$$\begin{array}{rcl} A^- & ::= & \Uparrow A^- \\ V^- & ::= & \mu\Uparrow a^+.c \\ S^- & ::= & \Uparrow \cdot S^+ \end{array}$$

$$\frac{c : (\Gamma \vdash a^+ : A^+)}{\Gamma \vdash \mu\Uparrow a^+.c : \Uparrow A^+}\ \textbf{(thunk-R)} \qquad \frac{\Gamma \mid S^+ : A^+ \vdash \Delta}{\Gamma \mid \Uparrow \cdot S^+ : \Uparrow A^- \vdash \Delta}\ \textbf{(thunk-L)}$$

$$\langle \mu\Uparrow(a).c \parallel \Uparrow \cdot S \rangle \triangleright c[S/a] \qquad \textbf{(thunk)}$$

$$\mu\Uparrow a.\langle t \parallel \Uparrow \cdot a \rangle \triangleright^t t \qquad \textbf{(eta-thunk)}$$

We can now close our discussion of operational semantics and abstract machines by presenting how closures and thunks are used within CBPV semantics to switch between CBV and CBN in a syntax-determined manner.

Summing up System-**L** embeds both a CBV and CBN machine, each delimited by polarities: positive values/stack can only contain sub-values/sub-stacks of the same polarity, which means that by default, the strict/lazy aspect of its operational semantics is preserved during evaluation. Closures and thunks allow shifting between CBV and CBN semantics by enclosing a negative value inside a positive one, and a positive stacks in a negative one. This is the essence of Call-by-Push-Value (CBPV) semantics for functional languages.

In the System-**L**, polarities determine strict/lazy semantics by deciding which side takes control when reducing the critical pair $\langle \mu a.c \parallel \mu x.c' \rangle$: CBV has the left side take control, and CBN the right side. The shifts allow this behavior be flipped by changing the polarity on demand.

4.8 Interlude:Focusing and deterministic machines (2/2)

System-**L** is a fully polarized abstract machine, that covers all connectives of intuitionistic linear logic ($\otimes$,$\oplus$,&,$\multimap$) and their units. The two sorts of type, positive and negative, respectively give CBV and CBN semantics. They can furthermore be combined with closures and shifts to provide a mix of the two in the form of *polarized semantics*, which corresponds to the reduction rules introduced in this chapter.

Those semantics match up with those of focused intuitionistic linear logic. In fact, when removing the term-level syntax to only keep the typing sequent, every typing rule of System-**L** lines up with either a rule of the logical system, or with a no-op, making System-**L** a *term-language* for focused intuitionistic linear logic. In this system, types do not only describe

a collection of set-theoretic values, but also evaluation semantics. Consider for example pattern-matching pairs $A \otimes B$ and projection pairs $A\&B$. While denotations of both types can be taken in the same Cartesian product of sets, the expressions of those two types have different semantics at runtime (CBV v. CBN), which is tracked at type-level.

Example: From CBN to CBV with closures We now present a first example of shifting, in which a negative command (therefore with CBN semantics) can be forced to have CBV semantics by encapsulating the top-level term in a closure. We will again color commands in red to emphasize their relative evaluation order. Let c_V and c_S be two commands such that:

$$c_V \triangleright^* \langle V \parallel a \rangle^-$$
$$c_S \triangleright^* \langle x \parallel S \rangle^-.$$

Then, the term $\mu a^-.c_V$ can be said to eventually return V^-, since $\mu a^-.\langle V \parallel a\rangle^-$ eta-reduces to V. Likewise, $\mu x.c_S$ will be said to return continuation S.

We shall use the fact that if σ is a substitution, and $c \triangleright c'$, then $c\sigma \triangleright c'\sigma$. The whole program $\langle \mu a.c_V \parallel \mu x.c_S\rangle$ runs as shown below, with c_S being evaluated before c_V, as expected of negative commands with CBN semantics.

$$\begin{aligned}
&\langle \mu a.c_V \parallel \mu x.c_S\rangle^- \\
\triangleright\ & c_S[(\mu a.c_V)/x] \\
\triangleright^*\ & \langle x \parallel S\rangle[(\mu a.c_V)/x] = \langle \mu a.c_V \parallel S\rangle \\
\triangleright\ & c_V[S/a] \\
\triangleright^*\ & \langle V \parallel a\rangle[S/a] = \langle V \parallel S\rangle^-
\end{aligned}$$

The command defining the value V cannot be run before its continuation, it is only run at its point of use (i.e. when interacting with S). This may be a problem: for example, if V is a function and c_V is the command defining it, this means the function cannot be define before it is called! Closures override this behavior and assign CBV semantics to function, and computations in general. To this end, consider now commands $c_{\Downarrow V}$ and $c_{\Downarrow S}$ such that:

$$c_{\Downarrow V} \triangleright^* \langle \Downarrow V \parallel a\rangle^+$$
$$c_{\Downarrow S} \triangleright^* \langle y \parallel \mu\Downarrow x.\langle x \parallel S\rangle^-\rangle^+$$

This is the same situation as before, but now V is wrapped in a (positive) closure that the continuation unwraps before passing V to S. The program now runs as show below, with the command $c_{\Downarrow V}$ running first, eventually returning the closed V, then $c_{\Downarrow S}$ runs, eventually performing the actual call. This is indeed the expected behavior of closures over functions in CBV languages: the term evaluating the closure has priority over the environment using it.

$$
\begin{aligned}
& \langle \mu a.c_{\Downarrow V} \parallel \mu y.c_{\Downarrow S} \rangle^{+} \\
\triangleright \;\; & c_{\Downarrow V}[(\mu y.c_{\Downarrow S})/a] \\
\triangleright^{*} \;\; & \langle \Downarrow V \parallel a \rangle[(\mu y.c_{\Downarrow S})/a] = \langle \Downarrow V \parallel \mu y.c_{\Downarrow S} \rangle \\
\triangleright \;\; & c_{\Downarrow S}[\Downarrow V/x] \\
\triangleright^{*} \;\; & \langle y \parallel \mu \Downarrow x.\langle x \parallel S \rangle \rangle[\Downarrow V/y] = \langle \Downarrow V \parallel \mu \Downarrow y.\langle y \parallel S \rangle \rangle \\
\triangleright \;\; & \langle V \parallel S \rangle
\end{aligned}
$$

Example: CBV to CBN with thunks Thunks override the CBV semantics of positive terms. This allows us to delay the evaluation of some data until a specified point of a program. Without thunks, positive terms are always evaluated at their point of definition. Thunks, or the other hand, are not evaluated, until an evaluation context requests this evaluation (it *forces* the thunk). To show this, we shall proceed the same way as for closures, skipping a little exposition for brevity.

Consider a command c_V defining a positive value V, and another c_S consuming that value through a stack S, namely:

$$
\begin{aligned}
c_V & \triangleright^{*} \langle V \parallel a \rangle^{+} \\
c_S & \triangleright^{*} \langle x \parallel S \rangle^{+}
\end{aligned}
$$

As shown below, polarity has the evaluation of c_V take priority over that of c_S when both are put together:

$$
\begin{aligned}
& \langle \mu a.c_V \parallel \mu x.c_S \rangle^{+} \\
\triangleright \;\; & c_V[(\mu x.c_S)/a^{+}] \\
\triangleright^{*} \;\; & \langle V^{+} \parallel a^{+} \rangle[(\mu x^{+}.c_S)/a] = \langle V^{+} \parallel \mu x^{+}.c_S \rangle \\
\triangleright \;\; & c_S[V^{+}/x^{+}] \\
\triangleright^{*} \;\; & \langle x^{+} \parallel S^{+} \rangle[V^{+}/x^{+}] = \langle V \parallel S \rangle^{+}
\end{aligned}
$$

Let us now embed this value inside a thunk: we have a command $c_{\Uparrow V}$ eventually producing a thunk $\mu\Uparrow b.\langle V \parallel b\rangle$ and a second, $c_{\Uparrow S}$, which eventually produces a stack $\Uparrow \cdot S$ forcing this thunk:

$$c_{\Uparrow V} \triangleright^* \langle \mu\Uparrow b^+.\langle V \parallel b\rangle^+ \parallel a^-\rangle$$
$$c_{\Uparrow S} \triangleright^* \langle x^- \parallel \Uparrow \cdot S^+\rangle$$

The whole programs now runs $c_{\Uparrow S}$ before $c_{\Uparrow V}$: the continuation executes before the thunked value, giving the intended semantics to thunks:

$$\begin{aligned}
&\langle \mu a.c_{\Uparrow V} \parallel \mu x.c_{\Uparrow S}\rangle^- \\
\triangleright\ & c_{\Uparrow S}[(\mu a^-.c_{\Uparrow V})/x^-] \\
\triangleright^*\ & \langle x \parallel \Uparrow \cdot S\rangle^-[(\mu a^-.c_{\Uparrow V})/x^-] = \langle \mu a.c_{\Uparrow V} \parallel \Uparrow \cdot S\rangle^- \\
\triangleright\ & c_{\Uparrow V}[(\Uparrow \cdot S^+)/a^-] \\
\triangleright^*\ & \langle \mu\Uparrow b.\langle V \parallel b\rangle^+ \parallel a\rangle^-[(\Uparrow \cdot S^+)/a^-] = \langle \mu(\Uparrow b).\langle V \parallel b\rangle^+ \parallel \Uparrow \cdot S\rangle^- \\
\triangleright\ & \langle V \parallel S\rangle^+
\end{aligned}$$

Example: CBV curried functions In System-**L**, functions have CBN semantics: this means that the expression defining a function is not evaluated before that function is called. Furthermore, they cannot be curried. The Call-by-push-value semantics of **L** nevertheless enable currying functions with CBV semantics to have a systematic encoding. To begin, note that System-**L** does have an equivalence between three basic functions types:

$$(A_1^+, \cdots, A_n^+) \multimap B^- \not\equiv (A_1^+ \otimes \cdots \otimes A_n^+) \multimap B^- \not\equiv A_1^+ \multimap \cdots \multimap A_n^+ \multimap B^-$$

Nevertheless, even functions of the latter type cannot be curried in the sense of CBV. A two-argument function $f : A^+ \multimap B^+ \multimap C^-$ can be called with an argument $x : A$, giving a term $\mu a.\langle f \parallel x \cdot a\rangle$. But this term isn't a value, and should we want to bind it as g in a continuation $\mu g.c$, the reduction of the machine goes as

$$\langle \mu a.\langle f \parallel x \cdot a\rangle \parallel \mu g.c\rangle^- \triangleright c[\mu a.\langle f \parallel x \cdot a\rangle/g].$$

As we can see, the call isn't evaluated before the resulting computation is passed to the continuation: partial application is not complete. We can force CBV semantics and complete

this partial evaluation functions by transforming the type $A \overset{\text{CBV}}{\to} B$ of CBV functions into $\Downarrow A \multimap \Uparrow B$. Applied to our function f, this gives

$$f : \Downarrow A \multimap \Uparrow\Downarrow B \multimap \Uparrow C.$$

With this new type, an intermediate closure over a function can be captured by a continuation after the argument $x : A$ is passed and the first of the two application complete: in the same setting as before, reduction now goes as follows, with V the intermediate closure:

$$\begin{aligned}
&\langle \mu a.\langle f \parallel \mu\Downarrow f'.\langle f' \parallel x \cdot \Uparrow \cdot \mu a\rangle\rangle \parallel \mu g.c\rangle \\
&\triangleright \langle f \parallel \mu\Downarrow f'.\langle f' \parallel x \cdot \Uparrow \cdot \mu\mu g.c\rangle\rangle \\
&\triangleright \langle f \parallel x \cdot \Uparrow \cdot \mu g.c\rangle \\
&\triangleright {}^*c[V/g]
\end{aligned}$$

Introducing intermediate closure and thunk around a function therefore gives it CBV semantics (thanks to the intermediate thunk), and allows it to be passed around (thanks to the intermediate closure). We will see later that this scheme extends to one compiling CBV λ-calculus, and furthermore ML-style programming languages, to System**L** by introducing new thunks and closure as to make the type of every compiled λ-term positive. Another such scheme turns all those types negative, endowing languages *à la* ML with call-by-need semantics, implemented using the other double arrow $\Downarrow\Uparrow$.

We shall furthermore use the two shifts to enrich programs with explicit resource manipulations which match their operational semantics: in positive mode, control and resources flow towards the left side, and vice versa in negative mode. Closures and thunks will serve as control points in which resources are automatically passed between the two different sides: from towards the left in CBV mode, and towards the right in CBN mode. We can furthermore account for this change at type-level, allowing us to track the dynamics of resources manipulation at compile-time within using the type system.

Before moving on, we do take the time present the *structural block* of the machine: constructs for sharing and for recursive computation, with are the only constructs in the System-**L** which aren't linear.

4.9 Sharing and fixpoints: the structural block

System-**L** can accommodate sub-structural type systems, which we will use to represent resource-flow within programs. This requires limiting the use of structural rules in such a way that only values of particular types can occur in a non-linear substitution (those are substitution where terms are replaced zero or more than one time). This is achieved by restricting the rules of *weakening* (not using a variable) and contraction (duplicating a variable) within the type system.

Amongst the many possible choices of sub-structural type systems, we implement intuitionistic linear logic with its *exponential modality*. Exponential types have weakening and contraction, while all others do not. We have made this choice as to guarantee at the same time that resources are preserved by program evaluation (linear), and data is freely shareable as in high-level functional languages (exponential).

Sharing The exponential modality of linear logic is implemented by new value/stack pair with a positive type, written $!A$ for an inner value of type A^-. They respect the following rules: (1) only values with all free variables of exponential types may be *promoted* to exponential values; (2) exponential values may be *weakened* and *contracted*; and (3) they might be *demoted* back to the underlying value.

Weakening and contractions are usually structural rules, which involve no term-level syntax. Instead, we create a new *structural* command which can implement this rule at term-level. This command is written $\langle \sigma; c \rangle$ where c is another command and σ a (non-linear) substitution of variables for variables. Those substitutions may take the form $[/x]$, which signifies that the value V is weakened (i.e. unused in c). Typechecking ensures that all other runtime substitutions in **L** are linear.

The command $\langle \sigma; c \rangle$ has a neutral polarity, and is reduced by non-linearly substituting the zero-one-or-more variables in σ and then proceeding to c. All variables being substituted within σ must have exponential types, which furthermore gives their types to the fresh variable

they substitute for.

$$A^+ ::= !A^-$$
$$\sigma ::= [] \mid \sigma[x^+/y^+] \mid \sigma[/x^+]$$
$$c ::= \langle \sigma ; c \rangle$$

$$\langle \sigma ; c \rangle \triangleright c\sigma \qquad \textbf{(struct)}$$

*(no eta-law for **(struct)**)*

$$\frac{c : (\Gamma', \overrightarrow{y_i : A_i} \vdash \Delta) \qquad \overrightarrow{!\Gamma' \vdash y_i : !A_i} \qquad \overrightarrow{!\Gamma' \vdash z_j : !B_j}}{\langle \overrightarrow{[y_i/x_i]}[\overrightarrow{/z_j}]; c \rangle : (\Gamma, !\Gamma' \vdash \Delta)} \textbf{(struct)}$$

Promotion and demotion Promotion and demotion respectively pack and unpack the inner value held within an exponential. Note that demotion can cancel promotion: if the inner value is packed then unpacked, nothing is changed. On the other hand, an exponential of type $!A$, once demoted to an A, cannot be promoted another time, as A is not a linear type. This means we should have an η-law for introducing exponential values, but not for eliminating them. As such, the binder on which the η-law applies must be on the value side, not on the stack that consumes it, a departure from other positive types. This peculiarity put aside, exponentials are straightforwardly implemented. The value $\mu!a^-.c$ promotes the output of the command c to an exponential value, provided in only depends on other exponential variables, and the stack $! \cdot S^-$ demotes it to feed the underlying value to S^-:

$$V^+ ::= \mu!a^-.c \quad S^+ ::= ! \cdot S^-$$

$$\langle \mu!a.c \parallel ! \cdot S \rangle \triangleright x[S/a] \qquad \textbf{(exp)}$$
$$\mu!a.\langle t^+ \parallel ! \cdot a \rangle \triangleright^t t^+ \qquad \textbf{(eta-exp)}$$
$$\frac{c : (!\Gamma \vdash a : A^-)}{!\Gamma \vdash \mu!a.c : !A^-} \textbf{(promotion)} \qquad \frac{\Gamma \mid S : A^- \vdash \Delta}{\Gamma \mid ! \cdot S : !A^- \vdash \Delta} \textbf{(demotion)}$$

To sum things up: values in the System-**L** are linear, and exponentials allow this constraint to be lifted. We shall use this to encode ML-style languages, using explicit scope management being used track the potential given to variables. As opposed to Hoffmann's *Resource-Aware ML*, we will use linearity to simplify the workings of resource preservation. Using linear logic with the exponential modality allow those two types of entity, data and resources, to cohabit.

Recursion Finally, we introduce recursive computations as a fundamental building block on which to build algorithms. We do not seek to compute the complexity of arbitrary fixpoints, for two reasons: (1) the problem is Turing-complete, which would doom our efforts, leading to partial solutions with many pitfalls, and (2) the AARA methods all work through some kind of *structural induction* on data structures. Nevertheless, simple fixpoints, such as those structural inductions, are perfectly analyzable, and good theoretical behavior are preserved by this addition.

Therefore, we choose to include fixpoints on our formalism, while controlling their unfolding and proving that resources are preserved at every step of small-step reduction. We shall use those fixpoints to define the usual iterators on algebraic data-types, and recover the precision on those constructions provided by AARA.

Our encoding and study of recursive computations in System-**L** is, in our understanding, novel. It mimics the well-known encoding of recursions in the λ-calculus using fixpoints combinators. Recall that, in the untyped λ-calculus, a term $\mathbf{Y}$ exists such that for any f, $\mathbf{Y}f$ reduces to $f(\mathbf{Y}f)$. Then, if f is implementable in a typed λ-calculus as a function $f = \lambda f'.\lambda x.t : (A \to B) \to A \to B$ then $\mathbf{Y}f$ can be considered a recursive function, namely the result of recursively calling f as f' within its body. While $\mathbf{Y}$ is not generally definable in a typed setting, it can be added as a primitive that, given a $t : A \to A$, returns a fixpoint $\mathbf{Y}t : A$ such that $\mathbf{Y}t \simeq t(\mathbf{Y}t)$.

This means fixpoints are terms that are stable under a kind of continuation: a fixpoint is defined by a functional t that preserves it. Skipping over some technicalities, they are characterized as solutions to equations of the type $x \simeq tx$. It is therefore natural to translate fixpoints to the System-**L** such that, for every S with continuation variable a, there is a fixpoint V such that $V \simeq \mu a.\langle V \parallel S\rangle$. This is the key to our encoding.

We need to add two alterations to obtain a construct suitable to our needs: (1) fixpoints should produce exponential values (otherwise they either don't use themselves and aren't recursive, or do consume themselves and cannot be returned), and (2) they should not be directly evaluate to their unfolded equivalent, as to preserve our small-step operational semantics. The first alteration is easily implemented by making fixpoints have type $!\mathbf{fix}A$, and the second one realized by guarding fixpoint unfolding with a constructor, forcing it to take a step of evaluation.

Formally, the stack constructor $\mathbf{fix} \cdot S$ unfolds one level of a fixpoint and feeds the resulting computation to S. On the other side, fixpoints have syntax $\mu\mathbf{fix}(a).\langle \mathbf{self} \parallel S \rangle$, derived from the syntax $\mu!a.c$ of exponentials. Note that **self** is not an identifier, it instead is part of the syntax, evocative of a hole to be filled with the fixpoint itself during unfolding. The continuation S within the value acts as the defining continuation of the fixpoint, making them satisfy the equation $V \simeq \mu a.\langle V \parallel S \rangle$. Their syntax is given below:

$$\begin{aligned} A^- &::= \mathbf{fix}\, A^- \\ V^- &::= \mu\mathbf{fix}(a).\langle \mathbf{self} \parallel S^- \rangle \\ S^- &::= \mathbf{fix} \cdot S^- \end{aligned}$$

The accompanying reduction rule **(fix)**, shown below, moves the entire fixpoint to the left side of a new command, copies S, feeds V to it, and passes the entire thing to the evaluation context. Note that an implicit α-conversion occurs when applying this rule, which protects the copies of S from unwanted substitution.

$$\langle \mu\mathbf{fix}(a).\langle \mathbf{self} \parallel S \rangle \parallel \mathbf{fix} \cdot S' \rangle \triangleright \langle \mu\mathbf{fix}(a).\langle \mathbf{self} \parallel S \rangle \parallel S \rangle [S'/a] \qquad \textbf{(fix)}$$

*(no eta-law for **(fix)**)*

Finally, here are the typing rules for fixpoints. They are also directly inspired by those for exponentials.

$$\frac{!\Gamma \mid S : !\mathbf{fix}\, A^- \vdash a : A^-}{!\Gamma \vdash \mu\mathbf{fix}(a).\langle \mathbf{self} \parallel S \rangle : !\mathbf{fix}\, A^-}\ \textbf{(fix-V)} \qquad \frac{\Gamma \mid S^- : A^- \vdash \Delta}{\Gamma \mid \mathbf{fix} \cdot S^- : \mathbf{fix} A^- \vdash \Delta}\ \textbf{(fix-S)}$$

4.10 Closing remarks

We have presented a CBPV machine that subsumes the CBV and CBN machine. Control flow and data flow are made explicit by this extension: data, control and resources pass the $\parallel$ mark of the top-level command at each reduction step, left-to-right in positive mode, and right-to-left in negative mode. The machine enjoys a factorized type system: the identity fragment deals with control flow, the logic fragment deals with data and computation, the structural fragment deals with the non-linear aspects of scope management, including recursion. This explicitation of control/data/resource flow is translated at type-level: if an expression has type $\Downarrow(!A) \multimap \Uparrow B$,

then we know it evaluates to a linear CBV function, that then can be transported "out of its scope of definition", then opened to give a computation. This computation takes a shared data/closure argument of type A and then defines another computation, that can finally be forced later "within its scope of use" to produce a linear data/closure of type B.

Tracking the dynamics of program evaluation at type-level in such a way is key to our resource analysis. This makes the System-**L** a relevant intermediate representation for resource analysis: once compiled into the machine, the operational semantics of a host language are made explicit. This allows us to build a systematic resource analysis for different programming languages, with increasingly more features. For example, in chapter 5, we will compile an ML-style language into the machine for analysis, and will then extend this workflow with monadic effects, monad transformers, and a do notation, all of this without changing the compilation/analysis of the core language. The factorization of concerns brought on by the machine will let us define this reusable resource analysis by orthogonal components: resource-flow will be added only by rewriting the identity fragment of programs to-and-from the machine, static analysis for size, complexity, and resource quantity will be added only by extending the logic fragment (with first-order constraints), potential will be added only by changing the structural fragment, and finally AARA-style indices for amortized complexity will only require extending data/computation type definitions with constraints.

Chapter 5

Frontend : from *ML* to System-**L**

System**L** is a capable intermediate representation for functional languages. On top of it, we build a frontend consisting of an ML-style functional programming language (CBV semantics, non-linear, not continuation-passing). This "Mini-ML" is representative of the core of ML-style languages, which are a lot closer to programmer concerns than the virtual machine. In between those two representations sits a CBPV ML-style language, with linear types, which serves as a convenient language to write libraries and advanced features such as monads and monad transformers. Those advanced features can then be compiled into System-**L**, allowing for resource analyses to be used on larger languages without having to extend the core analyzer.

5.1 A Mini-ML

The frontend for our implemented analyzer in a typed, functional programming language with CBV semantics *à la* ML. It includes algebraic data-types with case analysis and (mutually recursive) functions. Its syntax is given below in figure 5.1 and repeated in the appendix.

We use suspension dots "..." informally to denote sequences (possibly with delimiters) and square brackets to denote optional elements. In what follows, x, f, g denote a variable identifier, k a constructor identifier, A a type variable identifier, and finally K a type constructor identifier.

This language is given the well-known simple type system in line with the ML family of languages. Type definitions are all implicitly mutually recursive, and extend over the entire program, including expression which appear before said definitions. We assume all top-level

⟨*program*⟩ ::= ⟨*toplevel*⟩ ...

⟨*toplevel*⟩ ::= ⟨*def*⟩ | ⟨*expr*⟩

⟨*def*⟩ ::= `type (`[*A* `,`...`])` *K*`=` ⟨*consdef*⟩ [...]
| `let` *x* `=` ⟨*expr*⟩
| `let rec` *f x* ...`=` ⟨*expr*⟩ [`and` ...]

⟨*consdef*⟩ ::= `|` *k* [`of` ⟨*type*⟩ `*` ...]

⟨*type*⟩ ::= *A*
| ⟨*type*⟩ → ⟨*type*⟩
| *K* [`(` ⟨*type*⟩ `,` ...`)`]

⟨*val*⟩ ::= ⟨*intlitt*⟩
| *x*
| *k* [`(` ⟨*val*⟩ `,` ...`)`]
| `fun` *x* `->` ⟨*expr*⟩
| `rec` *x* `=` ⟨*val*⟩ [`and` ...] `in` ⟨*val*⟩

⟨*intlitt*⟩ ::= `0` | `1` | `2` | ...

⟨*intop*⟩ ::= `add` | `sub` | `eq` | `le` | ...

⟨*expr*⟩ ::= `x`
| ⟨*intlitt*⟩
| ⟨*intop*⟩
| ⟨*expr*⟩ `:` ⟨*type*⟩
| *k* [`(` ⟨*expr*⟩ `,` ...`)`]
| `match` ⟨*val*⟩ `with` ⟨*clause*⟩ [...] `end`
| `fun` *x* `->` ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*expr*⟩
| `let` *x* `=` ⟨*expr*⟩ `in` ⟨*expr*⟩
| `let rec` *x* `=` ⟨*val*⟩ [`and` ...] `in` ⟨*expr*⟩
| `tick` ⟨*intlitt*⟩ `in` ⟨*expr*⟩

⟨*clause*⟩ ::= `|` *k* [`(` *x* `,` ...`)`] `->` ⟨*expr*⟩

Figure 5.1: Mini-ML syntax

identifiers for variables, types and constructors are unique. Lastly, all recursive definitions must define functions.

In Mini-ML, values are either an integer literal, a constructor application, a closure containing a function, or a recursive closure. The language also includes a `tick` *k* `in` *e* expression which acts as the *tick* construct. The standard head-reduction, call-by-value, big-step semantics for this ML language assumed. This frontend language, suitable for functional programmers without with a low cognitive overhead, is compiled first to a Call-by-push-value ML, then to System-**L**.

5.2 A Call-by-push-value ML

The Call-by-push-value ML (CBPV-ML) used to compile the frontend language is defined below. This intermediate target enables addition to the frontend language without having to work directly with the machine representation. This language is adapted from Levy's original Call-by-push-value λ-calculus originally introduced by Levy in [54] and expanded in [53]. It was furthermore the topic of Levy's PhD thesis, which also gives a treatment of effects [52].

⟨*program*⟩ ::= ⟨*toplevel*⟩ . . . ⟨*expr*⟩

⟨*toplevel*⟩ ::=
| `data` K`[(`A`:±,...)] = [`⟨*consdef*⟩ . . . `]`
| `comput` K`[(`A`:± , ...)] = [`⟨*methdef*⟩ . . . `]`
| `type` K `[(`A `: ± , ...)] =` ⟨*type*⟩
| `let` x `=` ⟨*expr*⟩

⟨*consdef*⟩ ::= `|` k`[(`⟨*type*⟩`,...)]`

⟨*methdef*⟩ ::= `|` k`[(`⟨*type*⟩`,...)] ->` ⟨*type*⟩

⟨*type*⟩ ::= A | K`[(`⟨*type*⟩`,...)]` | `!` ⟨*type*⟩

⟨*intlitt*⟩ ::= `0` | `1` | `2` | . . .

⟨*intop*⟩ ::= `add` | `sub` | `eq` | `le` | . . .

⟨*clause*⟩ ::= `|` k `([`x`,...]) ->` ⟨*expr*⟩

⟨*val*⟩ ::= ⟨*intlitt*⟩
| x
| k`([`⟨*val*⟩`,...])`
| `closure` ⟨*expr*⟩
| `rec` x `=` ⟨*expr*⟩
| `exp` ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*intop*⟩
| ⟨*expr*⟩ `:` ⟨*type*⟩
| `get` ⟨*clause*⟩ . . . `end`
| ⟨*expr*⟩`.`k`([`⟨*val*⟩`,...])`
| `match` ⟨*val*⟩ `with` ⟨*clause*⟩ . . . `end`
| `let` x `=` ⟨*val*⟩ `in` ⟨*expr*⟩
| `tick` ⟨*intlitt*⟩ `in` ⟨*expr*⟩
| `thunk` ⟨*val*⟩
| `force` x `=` ⟨*expr*⟩ `in` ⟨*expr*⟩
| `open` ⟨*val*⟩
| `unfold` ⟨*val*⟩
| `unexp` ⟨*val*⟩

Figure 5.2: CBPV-ML syntax

Some syntactic adjustments were made to align its terminology with our own. Not only do it refine the CBV semantics of Mini-ML, but it also is linear, with implicit sharing of exponential types, inspired by Ehrhard [24]. Finally, recursive computations are implemented in the same style as in System-**L**: recursive values are all exponentials, and all exponentials are closures.

Syntax the syntax and small-step weak-head reduction of this CBPV-ML is given in figure 5.2. Its simple type system is sufficiently similar to the one of the machine that we consider it redundant. We also reuse the notations of System-**L** type for simplicity.

A CBPV-ML program is a set of declarations, followed by a single expression. Definitions introduce data-types, computation types, type synonyms, and values. Amongst those, type definitions are identical to those used in System-**L**.

Values in CBPV-ML are either an integer, a variable, a constructor application $k(\dots)$, or a closure (possibly exponential or recursive). As opposed to the matchine, recursive expressions in CBPV-ML are disjoint from values. Expressions are either integer primitives, computations

`get...end`, method calls, case-analysis on a constructor, a `let` binder, a cost `tick`, a thunk, the forcing of a thunk `force`, or the opening of a closure with `open`, `unexp`, or `unfold`. We use the shorthand `get` x `->` e for functions, $f.(x)$ for application, (V, W) for pairs, and $\iota_n(V)$ for sum types.

Reduction The CBPV-ML is equipped with a small-step operational semantics. It is defined the transitive closure of the basic reductions in figure 5.3, which sends expressions to expressions with an integer cost $k \in \mathbb{Z}$. The `let` binder substitutes a value as usual. Pattern-matching over constructors and methods is standard, and proceeds as presented for System-**L**. Costs are incurred with `tick` in the obvious way. Integer primitives are computations with a method `call`, which returns a thunk over its result. Opening closures with `open` moves control to the inner computation of the closure (the same goes for `unexp` and `unfold`). Lastly, forcing a thunk with `force` yield control to the thunk, which is reduced until it has the terminal form `thunk` V, at which point the produced value V can be substituted.

This lower-level representation combines the finer control of evaluation of CBPV with a lower cognitive overload when compared to System-**L**. Nevertheless, the two remain quite close: indeed, the System-**L** can be seen as merely the continuation-passing machine derived from this CBPV-ML.

5.3 Compilation from ML to CBPV-ML

Compiling the Mini-ML to CBPV is done according to the translation originally developed by Levy and cited above. In short, ML-expressions are transformed into CBPV-expressions, with ML-values becoming CBPV-values or thunks over those values, depending on context. A closure is used to provide CBV semantics to functions. We write $[\![-]\!]$ the compilation operator for expressions, $[\![-]\!]_{type}$ for the translation on type, and $[\![]\!]_{val}$ for the one on ML-values.

Types The type-level translation is transparent for all type constructor, except the function type, for which a closure and thunk are inserted. Exponentials are inserted throughout to

$$
\begin{array}{rll}
\texttt{let}\, x = V\, \texttt{in}\, e & \longrightarrow_0 & e[V/x] \\
\texttt{open}\, \texttt{closure}(e) & \longrightarrow_0 & e \\
\texttt{unfold}\, \texttt{rec}\, y = e & \longrightarrow_0 & e[(\texttt{rec}\, y = e)/y] \\
\texttt{force}\, x = e\, \texttt{in}\, e'' & \longrightarrow_k & \texttt{force}\, x = e'\, \texttt{in}\, e'' \qquad \textit{when } e \to_k e' \\
\texttt{force}\, x = \texttt{thunk}\, V\, \texttt{in}\, e & \longrightarrow_0 & e[V/x] \\
\texttt{unexp}\, \texttt{exp}\, e & \longrightarrow_0 & e \\
\texttt{tick}\, k\, \texttt{in}\, e & \longrightarrow_k & e \\
\texttt{match}\, k\, (\, V_1\, , \ldots , V_n\,)\, \texttt{with} & & \\
\quad \mid k\, (\, x_1\, , \ldots , x_n\,)\, \texttt{->}\, e & & \\
\quad \mid \ldots & \longrightarrow_0 & e[V_1/x_1, \ldots, V_n/x_n] \\
\texttt{end} & & \\
e.k(V_1,\ \ldots,\ V_n) & \longrightarrow_k & e'.k(V_1,\ \ldots,\ V_n) \qquad \textit{when } e \to_k e' \\
\textit{primop}\, .\texttt{call}(n,m,\ldots) & \longrightarrow_0 & \texttt{thunk}\, p \qquad \textit{(integer primitives)} \\
(\, \texttt{get} & & \\
\quad \mid k\, (\, x_1\, , \ldots , x_n\,)\, \texttt{->}\, e & & \\
\quad \mid \ldots & \longrightarrow_0 & e[V_1/x_1, \ldots, V_n/x_n] \\
\quad \texttt{end}\,)\, .\, k\, (\, V_1\, , \ldots , V_n\,) & &
\end{array}
$$

Figure 5.3: CBPV-ML reduction

allow well-typed Mini-ML programs to remain so when compiled to the linear CBPV-ML.

$$\begin{aligned}
[\![A]\!]_{type} &= !A \\
\left[\!\left[K(\vec{T})\right]\!\right]_{type} &= !K(\overrightarrow{[\![T]\!]_{type}}) \\
[\![T_1 \to T_2]\!]_{type} &= !\Downarrow([\![A]\!]_{type} \multimap \Uparrow[\![B]\!]_{type})
\end{aligned}$$

Top-level Whole programs are translated piece-wise, each toplevel definition in turn. First, a new type declaration *fun-decl* for functions is added. Then, one-by-one, type declarations are transformed into datatype definitions, and value definitions are translated. Mutually-recursive definitions get special treatment, which will be detailed later.

$$\begin{aligned}
\textit{fun-decl} &= \texttt{comput}\,(A:+) \to (B:-) = \texttt{call}\,(!A) \to B \\
\left[\!\left[\overrightarrow{decl}; e\right]\!\right]_{prog} &= \textit{fun-decl};\ \overrightarrow{[\![decl]\!]_{toplevel}};\ [\![e]\!] \\
\left[\!\left[\,\texttt{type}\,K(\vec{A}) = \overrightarrow{k(\vec{T})}\right]\!\right]_{toplevel} &= \texttt{data}\ K(\overrightarrow{A:+}) = \overrightarrow{k(\overrightarrow{[\![T]\!]_{type}})} \\
[\![\,\texttt{let}\,x = e]\!]_{toplevel} &= \texttt{let}\,x = (\,\texttt{let}\,y = \texttt{force}\,[\![e]\!]\,\texttt{in}\,\texttt{exp}\,y)
\end{aligned}$$

Values ML-values (integers, constructors, primitives, and functions) are translated to values in CBPV according to the following translation. It is transparent for all values except functions, which are wrapped in a closure to preserve their CBV semantics.

$$\begin{aligned}
[\![x]\!]_{val} &= x \\
[\![n]\!]_{val} &= n && \textit{(integer literal)} \\
[\![op]\!]_{val} &= \texttt{closure}\,op && \textit{(integer primitives)} \\
\left[\!\left[k(\vec{V_i})\right]\!\right]_{val} &= k\left(\texttt{exp}\left[\!\left[\vec{V_i}\right]\!\right]_{val}\right) \\
[\![\,\texttt{fun}\,x \to e]\!]_{val} &= \texttt{closure}\,(\,\texttt{get call}\,(x) \to [\![e]\!])
\end{aligned}$$

Expressions ML-expressions are translated into CPBV-expressions. ML-values are wrapped in a thunk, and all sub-expressions are forced before being used. This forcing guarantees CBV semantics of translated programs. Local definitions of recursive functions are treated

specifically once again.

$$
\begin{aligned}
[\![V]\!] &= \texttt{thunk exp}\,[\![V]\!]_{val} \\
[\![\,\texttt{let}\,x = e\,\texttt{in}\,e'\,]\!] &= \texttt{let}\,x = \texttt{force}\,[\![e]\!]\,\texttt{in}\,[\![e']\!] \\
[\![\,\texttt{tick}\,k\,\texttt{in}\,e]\!] &= \texttt{tick}\,k\,\texttt{in}\,[\![e]\!] \\
[\![k(\vec{e_i})]\!] &= \overrightarrow{\texttt{force}\,x_i = [\![e_i]\!]\,\texttt{in}}\ \texttt{thunk exp}\,[\![k(\vec{x_i})]\!]_{val} \\
\left[\!\left[\texttt{match}\,e\,\texttt{with}\,\overrightarrow{k(\vec{x}) \to e'}\right]\!\right] &= \texttt{force}\,y = [\![e]\!]\,\texttt{in} \\
&\quad\ \texttt{force}\,z = \texttt{unexp}\,[\![y]\!]\,\texttt{in} \\
&\quad\ \texttt{match}\,z\,\texttt{with}\,\overrightarrow{k(\vec{x}) \to [\![e']\!]} \\
[\![e_1\,e_2]\!] &= \texttt{force}\,x_1 = [\![e_1]\!]\,\texttt{in} \\
&\quad\ \texttt{force}\,x_2 = [\![e_2]\!]\,\texttt{in} \\
&\quad\ (\,\texttt{unexp}\,x_1).\,\texttt{call}\,(x_2)
\end{aligned}
$$

Recursion Finally, let us consider the case of mutually-recursive functions. ML has direct support for them, but CPBV only support one recursive value at a time. This causes no problems, as an intermediate computation can be created, which has each of the mutually-recursive functions as methods.

To perform this transformation for mutually-defined functions $\vec{x_i}$, the intermediate computation y is defined with methods method $\vec{k_i}$ encoding all recursive values $\vec{x_i}$. All instances of x_i within the definitions are accordingly replaced with a call to the k_i method of y. This allows y to be a simply-recursive value equivalent to all the x_i combined. Once y is defined, each x_i can be exported as a mere call to k_i on y.

To define this intermediate y, a fresh computation type definition *rec-decl*$(\vec{x_i})$ is created with methods $\vec{k_i}$. As to dispense with a type-inference step for the translation, the type A_i of each of x_i is taken as a parameter by the new type constructor, which allows translating the mutual recursions without knowing the types of the x_i in advance. This gives the implementation of both `let rec` constructs, local and top-level.

$$[\![\,\texttt{let rec}\,\overrightarrow{x_i = e_i}\,]\!]_{toplevel} = \begin{cases} \texttt{let}\, y = \left(\texttt{rec}\, y = \texttt{get}\, \left(\overrightarrow{k_i() \to [\![e_i]\!] \left[\overrightarrow{get_i(y)\,/\,x_i} \right]} \right) \right) \\ \overrightarrow{\texttt{let}\, x_i = get_i(y)} \end{cases}$$

$$[\![\,\texttt{let rec}\,\overrightarrow{x_i = e_i}\,\texttt{in}\, e'\,]\!] = \begin{cases} \texttt{let}\, y = \left(\texttt{rec}\, y = \texttt{get}\, \left(\overrightarrow{k_i() \to [\![e_i]\!] \left[\overrightarrow{get_i(y)\,/\,x_i} \right]} \right) \right) \,\texttt{in} \\ \overrightarrow{\texttt{let}\, x_i = get_i(y)}\,\texttt{in} \\ [\![e']\!] \end{cases}$$

$$get_i(y) = \texttt{exp}\,((\,\texttt{unfold}\, y).k_i())$$

$$\textit{rec-decl}(\overrightarrow{x_i}) = \texttt{comput}\, K(\overrightarrow{A_i} : -) = \overrightarrow{k_i() \to A_i}$$

Soundness This compilation scheme admits the following consistency theorems, which relates the cost-aware small-steps semantics of both languages:

1. Given an ML term e and with whose one-step reduction is $e \to e'$ (disregarding costs), then there is a sequence of small-step reductions $[\![e]\!] \to \cdots \to [\![e']\!]$ in CPBV-ML.
2. In the same situation as (1), when the single step has cost 0, (i.e. we have $e \to_0 e'$), then the reduction sequence in CBPV-ML has all zero costs as well, i.e. we have $[\![e]\!] \to_0 \cdots \to_0 [\![e']\!]$
3. In the same situation as (1), when the single step has cost k (i.e. we have $e \to_k e'$), then $e = \texttt{tick}\, k\, \texttt{in}\, e'$, and the reduction after translation is a single step of cost k. That is, we have:

$$\begin{array}{ccc} \texttt{tick}\, k\, \texttt{in}\, e' & \xrightarrow{\quad k \quad} & e' \\ \vdots & & \vdots \\ [\![-]\!] & & [\![-]\!] \\ \downarrow & & \downarrow \\ \texttt{tick}\, k\, \texttt{in}\, [\![e']\!] & \xrightarrow{\quad k \quad} & [\![e']\!] \end{array}$$

4. As a result of (2.) and (3.) the footprints of e and $[\![e]\!]$ are identical.

5.4 Compiling CPBV-ML to System-L

We have seen in the last chapter that the λ-calculus can be compiled to the CBV and CBN machine. Call-by-push-value λ-calculus can correspondingly be compiled into System-**L**. We now present this compilation scheme and its main properties regarding cost-aware reductions.

We still write $[\![-]\!]$ the compilation operator, this time from CBPV-ML to simply-typed System-**L**. Types and type definitions need no transformation between the ML-language and machine-language. The translation of CPBV-ML terms is presented in figure 5.4. The translation of values is literal, while the one of expression usually involves a continuation-capture followed by a command implementing the computation step. The η-reduction property in the System-**L** guarantees that those extraneous continuation captures can be safely removed. Lastly, the implicit structural rules of CBPV-ML are transformed into explicit commands as required in our presentation of System-**L**, but this explicitation of sharing is standard and elided from the figure for simplicity.

Top-level While type definitions need not be transformed between CBPV-ML and System-**L**, value definitions do. Those are compiled in two steps: first, all top-level `let` definitions are folded into a single expression with `let ...in` binder, and second, this single top-level entry point is compiled into the virtual machine.

This entry-point be given special treatment for technical reasons: indeed, CBPV-ML term are translated into System-**L** terms, but, in System-**L**, reduction is defined on *commands*. This is dealt with resolved by introducing a fresh continuation variable to serve as a "final continuation". Following already established convention in the literature, we write $\star$ this continuation variable, and translate the one expression that makes up the whole CBPV-ML program into a command $\langle [\![e]\!] \parallel \star \rangle$.

5.5 Results

1. Given a CBPV term e and a small-step reduction $e \to e'$ (disregarding costs), then there is a sequence of small-step reductions $[\![e]\!] \to \cdots \to [\![e']\!]$ in **L**.

2. In the same situation as (1), when the single step has cost 0, (i.e. we have $e \to_0 e'$), then the reduction sequence in **L** has all zero costs as well, i.e. we have $[\![e]\!] \to_0 \cdots \to_0 [\![e']\!]$

3. In the same situation as (1) when the single step has cost k (i.e. we have $e \to_k e'$), then $e = \texttt{tick}\, k\, \texttt{in}\, e'$, and the top-level reduction after translation is a two-step reduction

CBPV-ML Values $V \longmapsto$ System-L values $[\![V]\!]$

$$
\begin{aligned}
[\![x]\!]_{val} &= x \\
[\![n]\!]_{val} &= n \\
\left[\!\left[k(\vec{V})\right]\!\right]_{val} &= k(\overrightarrow{[\![V]\!]}) \\
[\![\,\texttt{closure}\,e]\!]_{val} &= \Downarrow[\![e]\!] \\
[\![\,\texttt{rec}\,x = e]\!]_{val} &= \mu(\mathbf{self} \cdot a).\langle \mathbf{self} \parallel \mu x.\langle [\![e]\!] \parallel a\rangle\rangle \\
[\![\,\texttt{exp}\,V]\!]_{val} &= \mu!a.\langle [\![V]\!] \parallel a\rangle
\end{aligned}
$$

CBPV-ML expressions $e \longmapsto$ System-L terms* $[\![e]\!]$

$$
\begin{aligned}
[\![\,\texttt{let}\,x = V\,\texttt{in}\,e]\!] &= \mu a.\langle [\![V]\!] \parallel \mu x.\langle [\![e]\!] \parallel a\rangle\rangle \\
[\![\,\texttt{tick}\,k\,\texttt{in}\,e]\!] &= \mu a.\langle \$k; \langle [\![e]\!] \parallel a\rangle\rangle \\
[\![\,\texttt{thunk}\,V]\!] &= \mu\Uparrow a.\langle [\![V]\!] \parallel a\rangle \\
[\![\,\texttt{force}\,x = e\,\texttt{in}\,e']\!] &= \mu a.\langle [\![e]\!] \parallel \Uparrow \cdot \mu x.\langle [\![e']\!] \parallel a\rangle\rangle \\
[\![\,\texttt{open}\,V]\!] &= \mu a.\langle [\![V]\!] \parallel \mu\Downarrow x.\langle x \parallel a\rangle\rangle \\
[\![\,\texttt{unfold}\,V]\!] &= \mu a.\langle [\![V]\!] \parallel \mathbf{fix} \cdot a\rangle \\
[\![\,\texttt{unexp}\,V]\!] &= \mu a.\langle [\![V]\!] \parallel\, ! \cdot a\rangle \\
[\![op]\!]_{val} &= op \\
[\![V]\!] &= [\![V]\!]_{val} \\
\left[\!\left[\,\texttt{get}\,\overrightarrow{k(\vec{x}) \to e}\,\texttt{end}\,\right]\!\right] &= \mu(\overrightarrow{k(\vec{x}; a).\langle [\![e]\!] \parallel a\rangle}) \\
\left[\!\left[e.k(\vec{V})\right]\!\right] &= \mu a.\langle [\![e]\!] \parallel k(\overrightarrow{[\![V]\!]}) \cdot a\rangle \\
\left[\!\left[\,\texttt{match}\,V\,\texttt{with}\,\overrightarrow{k(\vec{x}) \to e}\right]\!\right] &= \mu a.\langle [\![V]\!] \parallel \mu(\overrightarrow{k(\vec{x}).\langle [\![e]\!] \parallel a\rangle})\rangle
\end{aligned}
$$

**Compiling CBPV-ML expression also involves adding an explicit sharing $\langle \sigma\ ;\ \cdots \rangle$ to each newly created term, which is elided here for simplicity.*

Figure 5.4: Compilation from CBPV-ML to System-**L**

with the same cost profile:

$$\begin{array}{ccccc} \texttt{tick}\,k\,\texttt{in}\,e' & & \xrightarrow{\hspace{10em}}_{k} & & e' \\ \vdots\,[\![-]\!] & & & & \vdots\,[\![-]\!] \\ \langle \mu a.\langle \$k; \langle [\![e']\!] \parallel a\rangle\rangle \parallel \star\rangle & \longrightarrow_{0} & \langle \$k; \langle [\![e']\!] \parallel \star\rangle\rangle & \longrightarrow_{k} & \langle [\![e']\!] \parallel \star\rangle \end{array}$$

4. As a result of (2.) and (3.) the resource profiles, and thereby footprints, of the CBPV-expression e and command $\langle [\![e]\!] \parallel \star\rangle$ as are identical.

5.6 Implementing monad transformers in ML

As to extend the scope of our analysis system and present its extensibility, we now implement imperative blocks into Mini-ML. This extends the expressive power of input languages without increasing the complexity of the analysis. Those imperative constructs remain *local* in scope: that is, local mutable variables, iteration with early return are provided, but no global mutability or non-local control flow outside the imperative blocks is allowed. The implementation of those effects as monads and monad transformers within Call-by-Push-Value dispenses with nested closures, as opposed to an implementation native to Mini-ML. Let us begin with presenting the new primitives added to Mini-ML: monads[62] and monad transformers[55]. Let us briefly recall the basic definitions related those concepts.

A **monad** consists of a type constructor $M : \mathit{Type} \to \mathit{Type}$ endowed with functions $\mathit{bind}_M : MA \to (A \to MB) \to MB$ and $\mathit{return}_M : A \to MA$ subject to the following coherence laws:

$$\begin{aligned} \mathit{bind}\ x\ \mathit{return} &= x \\ \mathit{bind}\ (\mathit{return}\ x)\ f &= f\ x \\ \mathit{bind}\ (\mathit{bind}\ x\ f)\ g &= \mathit{bind}\ x\ (\texttt{fun}\ y \to \mathit{bind}\ (f\ y)\ g) \end{aligned}$$

A **monad transformer** consists of a type constructor $T : (\mathit{Type} \to \mathit{Type}) \to \mathit{Type} \to \mathit{Type}$, together with a monad structure on $T(M, -)$ whenever M is a monad. We also denote this monad as $T \circ M$. Furthermore, a monad transformer is endowed with a function $\mathit{lift}_T : MA \to T_M A$ subject to the following coherence laws:

$$\begin{aligned} \mathit{lift}_T\ \mathit{bind}_M &= \mathit{bind}_{T\circ M} \\ \mathit{lift}_T\ (\mathit{bind}_M\ x\ f) &= \mathit{bind}_{T\circ M}\ (\mathit{lift}_T\ x)\ (\mathit{lift}_T \circ f) \end{aligned}$$

Finally, a **runner** for a monad transformer T consists of a type constructor $R : \mathit{Type} \to \mathit{Type}$ together with a function $\mathit{run}_R : T(M, A) \to R(MA)$.

Identity, State, and Exception monads We provide one monad and two monad transformers in Mini-ML, the *identity* monad and the *state* and *exception* monad transformers. Those are abstract types in Mini-ML, and will be defined in CBPV-ML as negative types, not positive types as the rest of Mini-ML expressions. For reasons which shall become clear later, the primitives will be implemented as a family of functions indexed by the monad they implement. We encode then as macros, which can take expressions are parameters, but they can also be implemented as functions taking closures as arguments instead. The two implementations are equivalent in our setting, provided one runs a straightforward inlining of said functions and simplifies the resulting `open closure` e it induces.

The *identity* monad $I(-)$ merely provides CBN semantics to the computations encapsulated in it. It is endowed with a "run" functions which performs the computation and returns the computed value. The primitives of $I(-)$ are:

Identity monad $I(A)$

$$\begin{array}{rcl} \mathit{return}_I & : & A \to I(A) \\ \mathit{bind}_I & : & I(A) \to (A \to I(B)) \to I(B) \\ \mathit{runI} & : & I(A) \to A \end{array}$$

The *state* monad transformer $S_V(-,-)$ extends a monadic computation $M(A)$ with the ability to mutate a local variable of type V. It does so with primitives *get* and *set*, and provides a runner that demotes a stateful computation to a function taking an initial value for V as argument and returning a final state together with its result. The primitives of S_V are:

State monad transformer $S_V(M, A)$

$$\begin{array}{rcl} \mathit{return}_S & : & A \to S_V(M, A) \\ \mathit{bind}_S & : & S_V(M, A) \to (A \to S_V(M, B)) \to S_V(M, B) \\ \mathit{liftS} & : & M(A) \to S_V(M, A) \\ \mathit{runS} & : & S_V(M, A) \to V \to M(V \times A) \\ \mathit{get} & : & S_V(M, V) \\ \mathit{set} & : & V \to S_V(M, 1) \end{array}$$

The **exception** monad transformer $E_X(M, A)$ extends a monadic computation $M(A)$ with the possibility of throwing an exception of type X when run, which short-circuits the rest of

the computation. It has a *throw* primitive taking the exception X as a parameter. Its runner *runE* takes a computation of type $E_X(M, A)$, and returns a computation of type $M(X + A)$, that is, a computation which may return an exception. This is implemented with the following primitives:

Exception monad transformer $E_X(M, A)$

$$
\begin{array}{rcl}
\mathit{return}_E & : & A \to E_X(M, A) \\
\mathit{bind}_E & : & E_X(M, A) \to (A \to E_X(M, B)) \to E_X(M, B) \\
\mathit{liftE} & : & M(A) \to E_X(M, A) \\
\mathit{runE} & : & E_X(M, A) \to M(X + A) \\
\mathit{throw} & : & E \to E_X(M, A)
\end{array}
$$

Using those primitives, local imperative blocks can be added to the purely functional Mini-ML, while preserving the compilation scheme to CBPV-ML. First, let us see what those blocks look like. Then, we can proceed with the implementation of monads and monad transformers in CBPV-ML.

a do-notation for Mini-ML The payoff of implementing monad and transformers lies in the ability to port some imperative algorithms to our purely functional analysis setup through the use of a *do-notation*. This permits imperative blocks made up of statements, conditionals, and iterations with early-returns, *break*, and *continue* keywords. This imperative extension for purely functional languages is taken as-is from Ulllrich and De Moura's work on the *Lean* proof assistant [78, 77]. This syntax of imperative blocks is given below, and grouped into four families of statements that we shall present separately.

⟨*expr'*⟩ ::= `do` ⟨*stmt*⟩ | ⟨*expr*⟩

⟨*stmt*⟩ ::= `return` ⟨*expr*⟩
| `let` *x* `<-` ⟨*stmt*⟩ `;` ⟨*stmt*⟩
| `let` *x* `=` ⟨*expr*⟩ `;` ⟨*stmt*⟩

| `let mut` *x* `:=` ⟨*expr*⟩ `;` ⟨*stmt*⟩
| *x* `:=` ⟨*expr*⟩ `;` ⟨*stmt*⟩
| `if` ⟨*expr*⟩ `then` ⟨*stmt*⟩ [`else` ⟨*stmt*⟩] `end` `;` ⟨*stmt*⟩

| `return!` ⟨*expr*⟩

| `for` *x* `in` ⟨*expr*⟩ `do` ⟨*stmt*⟩ `done` `;` ⟨*stmt*⟩
| `continue!`
| `break!`

This extension technically splits the syntax into two sorts of expressions: pure expressions (*expr*) and possibly imperative expressions (*expr'*). Impure expressions may include imperative blocks, but blocks themselves cannot hold impure expressions. This implies imperative blocks cannot be nested. This separation can safely be ignored, but doing so induces some unusual semantics for nested blocks: a mutable variable declared in an outer block becomes non-mutable in its inner blocks, for example. We therefore choose to not allow it for simplicity.

We shall present the fours families of statements in fragments. Compilation-wise, imperative blocks are progressively translated into calls to the monadic primitives we introduced above. This compilation is done first for the last block, then the third, etc. until only ML-expressions and monadic primitives remain. the use of effects changes the monad used for the *bind* and *return* primitives in basic blocks, which is why *return* and *bind* are indexed by monads: the particular monad they implement will change depending on which effect a block uses.

Identity monad The first family comprises the basic monadic *do-notation*: an expression might be a `do` block containing a statement, which itself is a sequence of assignments terminated with a `return`. The `let` x `<-` b statement assigns the result of evaluating the effectful block b to x, while the `let` x `=` e statement is merely an assignment of the value of a pure expression. Those blocks are translated to the identity monad: `return` e is implemented with *returnI*, the effectful assignment with *bindI*, and finally the pure assignment via a combination of *bindI* and *returnI*. This translation scheme should be familiar to readers who are versed in programming with monads. Finally, the compilation of `do` b expressions wraps b with a call to *runI*, which provides the return value of the block.

Mutability with S_V The second family introduces the first proper effect: local mutable variables. The `let mut` binders declares a new mutable variable and initializes it. This variable's scope extends to the end of the block. Then, the assignment statement $x := e$ provides a new value for x. Finally, imperative conditionals `if ...then ...else` allow for mutations guarded by a boolean test. The `then` and `else` branches each bear a block returning a unit type, who can read and write mutable variables, and the assignments they make are still in effect in the continuing statement conditional block, as expected in imperative languages.

Mutability is compiled to the primitives of the S_V transformer. Each new declaration of a mutable variable in a `let mut` x `:=` e `;` b turns the block b of type $M(A)$ into a block of

type $S_V(A)$, where V is the type of e. The modified block is wrapped in a call to *runS* where the initial value of x is provided. This translation occurs for each declaration of a mutable variable, meaning a block with n mutable variables will eventually have a body whose monad is $S_{V_1} \circ \cdots \circ S_{V_n}$.

Access to the mutable variable x is implemented by binding a normal variable of the same name with a `let` x `<- get` binder. Each sub-block of b likewise incurs a re-binding of x through *get* after its evaluation, which shadows the old binding with one containing the freshest value, effectively resulting in a *single-static-assignment* form for the compiled block, up to α-equivalence. Assignments $x{:=}e$ are compiled to `set` e followed by further binding of x.

Early returns with E_X The third family of effects implements *early returns* from imperative blocks. This is used within branches of a `if` statement to bypass evaluation of the continuing statement of the conditional, as in this example:

```
let x <- b;
if x = 0 then return! e_early end;
...;
return e_normal
```

Here, the usual `return` statement of the first family of statements could not have been usedon the second line, as both blocks of conditionals must return unit types. The new `return!` statement instead has a polymorphic return type, as does the *throw* expressions in OCaml and Haskell. Typing ensures that the values of all returns (both final and early) match.

This third family of statement is compiled before the first two using the E_X transformer. First, the entire block is wrapped in a call to *runE* with acts a *catch* clause for when the early-returned value which might be thrown. Inside the block, early returns become *throw*, and each normal return is wrapped in a *LiftE* as to satisfy the required types.

Imperative iterations with E_X Finally, the last family of statement implements iteration over generic data structures (we implemented it for linked lists), together with the capacity to short circuit to the next iteration of the loop with `continue`, or to bypass the rest of the execution entirely with `break`, as is usual in imperative languages. The inner body of loops

are statements returning a unit type. In the case of nested loops, breaking and continuing only applies to the innermost loop.

This iterating structure requires a new primitive to fold over data structures, namely, for a data structure $T(A)$, we need a *foldM* function, reminiscent of list folds, which has type:

$$\mathit{foldM} : \forall M.\ (A \to B \to M(B)) \to T(A) \to B \to M(B)$$

Compilation of this final family in done before the other three, and in the same style as for early returns: each loop is wrapped in two *runE*: one outside the loop for `break` (called the outer one), and one inside for `continue` (the inner one). Since loop bodies are blocks $b : M(1)$ returning a unit type, this means the compiled bodies will have type $E_1(E_1 \circ M, 1)$.

Inside the body of the loop, `break` statement become $\mathit{throw}\ ()$ (which are caught by the outer *runE*), and `continue` statements become $\mathit{liftE}\ (\mathit{throw}\ ())$ (which are caught by the inner *runE*). This transformation is not applied to `break` and `continue` inside the bodies of nested loop, which are instead treated when the most nested loops are themselves compiled. Finally, final returns are lifted accordingly to their depth (two *liftE* per loop body they sit inside of).

Compilation of monad transformers (1/2) The monadic primitives of Mini-ML are implemented as typed macros in CBPV-ML, which allows for direct manipulations of expressions with negative types. First, monadic types are translated as follows, extending the translation of ML-types into CBPV-types:

$$\begin{aligned}
[\![I(A)]\!]_{type} &= \Uparrow [\![A]\!]_{type} \\
[\![S_V(M, A)]\!]_{type} &= \,!\,[\![V]\!]_{type} \multimap [\![M(V \times A)]\!]_{type} \\
[\![E_X(M, A)]\!]_{type} &= [\![M(X + A)]\!]_{type}
\end{aligned}$$

We can now present the implementation of primitives. This can only be done by inducting on the monads transformed monads, which means typing information is required to dispatch the correct implementation. First, runners are all trivial:

$$[\![\mathit{runI}]\!](e) = [\![\mathit{runS}]\!](e) = [\![\mathit{runE}]\!](e) = e$$

We can then implement the rest of the monads, starting with the identity monad. Its implementation as thunks also make its implementation trivial:

$$\begin{aligned}
[\![\mathit{return}_I]\!](e) &= e \\
[\![\mathit{bind}_I]\!](e_1, e_2) &= \texttt{force}\, x_1 = e_1 \,\texttt{in}\, e_2.(x_1)
\end{aligned}$$

The state monad transformer is implemented as a function taking a state, and returning a computation that will eventually produce a new state and a value. As such, its primitives all get take initial state s as argument, then call a primitive of the underlying monad with arguments that thread the state through the computation:

$$\begin{aligned}
[\![\mathit{return}_{(S_V \circ M)}]\!](e) &= \texttt{get}\, s \texttt{ -> } [\![\mathit{return}_M]\!](\texttt{force}\, x = e \,\texttt{in}\, \texttt{thunk}\, (s, x)) \\
[\![\mathit{bind}_{(S_V \circ M)}]\!](e_1, e_2) &= \texttt{get}\, s \texttt{ -> } [\![\mathit{bind}_M]\!](e_1.(s), \texttt{get}\, (s', x) \texttt{ -> } e_2.(x).(s')) \\
[\![\mathit{liftS}_{(S_V \circ M)}]\!](e) &= \texttt{get}\, s \texttt{ -> } [\![\mathit{bind}_M]\!](e, \texttt{get}\, x \texttt{ -> } [\![\mathit{return}_M]\!]((s, x))) \\
[\![\mathit{get}_{(S_V \circ M)}]\!] &= \texttt{get}\, s \texttt{ -> } [\![\mathit{return}_M]\!](\texttt{thunk}\, (s, s)) \\
[\![\mathit{set}_{(S_V \circ M)}]\!](e) &= \texttt{get}\, s \texttt{ -> } [\![\mathit{return}_M]\!](\texttt{force}\, s' = e \,\texttt{in}\, \texttt{thunk}\, (s', ()))
\end{aligned}$$

Finally, the exception monad transformer is implemented as a computation that returns a sum type. Its primitives are thereby implemented directly as call to the underlying monad, whose arguments may dispatch on the constructor on the sum type to potentially short-circuit the rest of the computation:

$$\begin{aligned}
[\![\mathit{return}_{(E_X \circ M)}]\!](e) &= [\![\mathit{return}_M]\!](\iota_2(e)) \\
[\![\mathit{bind}_{(E_X \circ M)}]\!](e_1, e_2) &= [\![\mathit{bind}_M]\!](e_1, e_{2'}) \\
\mathit{where}\ e'_2 &= \texttt{get}\, x \texttt{ -> } \texttt{match}\, x \,\texttt{with} \\
&\qquad \iota_1(\mathit{err}) \texttt{ -> } [\![\mathit{return}_M]\!](\texttt{thunk}\, \iota_1(\mathit{err})) \\
&\qquad \iota_2(y) \texttt{ -> } [\![\mathit{bind}_M]\!](\texttt{thunk}\, y, e_2) \\
&\quad \texttt{end} \\
[\![\mathit{liftE}_{(E_X \circ M)}]\!](e) &= [\![\mathit{bind}_M]\!](e, \texttt{get}\, x \texttt{ -> } [\![\mathit{return}_M]\!](\iota_2(x))) \\
[\![\mathit{throw}_{(E_X \circ M)}]\!](e) &= [\![\mathit{return}_M]\!](\iota_1(e))
\end{aligned}$$

This translation does not introduce any extraneous closures in the resulting CBPV-ML code, and, to our knowledge given empirical evidence, admits good simplification properties in the

presence of inlining and normalization-by-evaluation, leading eventually to direct-style, purely functional programs in System-**L** of comparable complexity to the original imperative block.

5.7 Closing remarks

With this frontend in place, now is a good time to stop and review our analysis setup. The input language is an ML-style, call-by-value programming language with algebraic data-types and recursive functions. It is extended with imperative blocks, which are implemented without incurring extraneous closures thanks to the implementation of monad and monad transformers in call-by-push-value.

After compilation into a Call-by-Push-Value ML language, the ordering of reduction steps is syntactically determined even in the presence of both CBV and CBN aspect in the source language. Then, compilation in System-**L** transforms ML-style CBPV programs into an abstract machine representation that provides first-class continuations. This does away with the notion of evaluation context, as reduction on the machine head suffices to represent weak-head reduction of CBV, CBN, and mixed style programming.

As far as resources are concerned, the resource profile of the original program is preserved by this compilation scheme, which means we now have access to it from System-**L**. This has two benefits. First, since the reductions ordering of the program is syntactically explicit, the ordering of costs also is, which is essential to a re-usable resource analysis. Second, since the machine does away with implicit evaluation context in the reduction rules, potential that may flow to-and-from this unknown context is no longer a problem.

This allows us to derive an analysis scheme on our intermediate representation that is simplified compared to similarly-powerful AARA implementations: following the explicit control flow in System-**L** programs, resources move through the program. When values and stack in the machine are substituted during head-reductions, the potential they hold can interact with those resources, which implement amortization. The linear type system we put over the abstract machine will be essential in implementing this flow of resources safely.

The rest of this manuscript describes how this analysis on System-**L** is designed and implemented. Starting with an extension to our linear type system, numerical annotations called parameters will be added to types allowing for sizes, data-structure shapes, amount

of resources, and potential to be approximated at type-level. Then, using a CPBV effect, a state-token will be passed along the flow of the program (using the explicit control flow of CBPV). Linearity will ensure that resources remain centralized as this token moves around, greatly reducing the amount of proof work required to show the soundness of our resource bounds.

Chapter 6

Extending the machine for static analysis

System-**L** forms the intermediate representation our resource analysis is based on, a choice motivated by its good theoretical properties and ability to syntactically encode operational semantics. For the analysis proper, we will extend the type system presented last chapter without altering those properties. Namely, reduction rules shall remain unchanged and no new term-level construct shall be added, except optional user-given annotations. The extended type machine then has an erasure procedure which is compatible with reduction.

The extension to the type system takes the form of a logical fragment at type level, which affect the typing procedure as to output, together with any well-typed program, a first-order constraint, whose free variables (called *parameters*) are exactly its free type-level variables. Any assignment of those parameters satisfying the constraints (called a *model*) gives program . An obvious application will be, for example, the size of a list or dynamically allocated array. We will also introduce resource manipulations, once again without effects on the reductions. The minimal model of the amount of those resources will be our safe resource bound. Finally, a compatible notion of potential will be introduced, which will enable AARA analysis.

6.1 First-order constraints

The first step of our work in this chapter is to create a general-purpose static analysis framework for System-**L**. We want this framework to be as "neutral" as possible, as to support many different theoretical frameworks, and also want to stay as close as possible to the core of the CHC, and use the fact that the System-**L** is a term language for sequent calculus to minimize

proof work. As such, we choose to design and implement this static analysis system as a fragment of multi-sorted, first-order logic, embedded at type-level in programs. This first section merely defines the usual machinery accompanying first-order logic, namely models and the intuitionistic sequent calculus LJ.

Models Given a first-order signature, and a set $\mathcal{C}$ of first-order sentences expressed with that signature called a *theory* (those sentences not necessarily taken from our limited fragment), a model $\mathcal{M}$ for that theory consists of an assignment, for each sort $\mathbf{s}$ of a set $[\mathbf{s}]$, for each operator φ of arity $(\vec{\mathbf{s}_i}) \to \mathbf{s}$ of a function $[\varphi] : \Pi_i[\mathbf{s}_i] \to [\mathbf{s}]$, and for each relation R of arity $(\vec{\mathbf{s}_i})$ a subset $[R]$ of $\Pi_i[\mathbf{s}_i]$.

A scope of variables $\Theta = \overrightarrow{\alpha : \mathbf{s}}$ is given by a function σ, written as a substitution, assigning to each α a value $\alpha\sigma \in [\mathbf{s}]$. We write $\Theta \models \sigma$ in such cases. This gives a model for terms $[\tau]_\sigma$ defined as $[\alpha]_\sigma = \alpha\sigma$ and $[\varphi(\vec{\tau})] = [\varphi](\overrightarrow{[\tau]})$. This allows us to define whether a model satisfies a constraint C, written $\sigma \models C$, using Tarski's "definition of truth"[76]. This is given by the following meta-theoretical rule system:

- $\sigma \models \top$ always, and $\sigma \models \bot$ never.
- $\sigma \models \tau = \tau'$ whenever $[\tau]_\sigma = [\tau']_\sigma$.
- $\sigma \models R(\vec{\tau})$ whenever $\overrightarrow{[\tau]_\sigma} \in [R]$
- $\sigma \models E$ whenever σ is a model of every factor in the conjunction E
- $\sigma \models E \wedge C$ whenever $\sigma \models E$ and $\sigma \models C$
- $\sigma \models E \Rightarrow C$ if either not $\sigma \models E$ or $\sigma \models C$
- $\sigma \models \exists \alpha : \mathbf{s}.C$ whenever there is some $x \in [\mathbf{s}]$ such that $\sigma[x/\alpha] \models C$
- $\sigma \models \forall \alpha : \mathbf{s}.C$ when for all $x \in [\mathbf{s}]$, we have $\sigma[x/\alpha] \models C$

Reasoning We restate here the fragment of sequent calculus relevant to our fragment of first-order logic and its soundness theorem. This will be useful in the rest of the chapter to embed constraints in our sequent-based type system. Given a theory, as a set of axioms

$\mathcal{C}$, the intuitionistic sequent-calculus LJ of Gentzen[25, 26], when restricted to the fragment $(\top, \bot, \wedge, \Rightarrow, \forall, \exists)$, is defined by the following rules.

$$\textbf{Identity:}\quad \frac{}{\Gamma, C \vdash C} \qquad \frac{C \in \mathcal{C}}{\Gamma \vdash C} \qquad \frac{\Gamma, C \vdash \Delta \qquad \Gamma \vdash C}{\Gamma \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma[\tau/\alpha] \vdash \Delta[\tau/\alpha]}$$

$$\textbf{Structure:}\quad \frac{\Gamma \vdash \Delta}{\Gamma, \Gamma' \vdash \Delta} \qquad \frac{\Gamma, \Gamma', \Gamma' \vdash \Delta}{\Gamma, \Gamma' \vdash \Delta}$$

$$\textbf{Logic } (\top/\bot)\textbf{:}\quad \frac{\Gamma \vdash \Delta}{\Gamma, \top \vdash \Delta} \qquad \frac{}{\Gamma \vdash \top} \qquad \frac{}{\Gamma, \bot \vdash \Delta}$$

$$\textbf{Logic } (\wedge)\textbf{:}\quad \frac{\Gamma, C, D \vdash \Delta}{\Gamma, C \wedge D \vdash \Delta} \qquad \frac{\Gamma \vdash C \qquad \Gamma \vdash D}{\Gamma \vdash C \wedge D}$$

$$\textbf{Logic } (\Rightarrow)\textbf{:}\quad \frac{\Gamma \vdash C \qquad \Gamma, D \vdash \Delta}{\Gamma, C \Rightarrow D \vdash \Delta} \qquad \frac{\Gamma, C \vdash D}{\Gamma \vdash C \Rightarrow D}$$

$$\textbf{Logic } (\forall)^1\textbf{:}\quad \frac{\Gamma C[\tau/\alpha] \vdash \Delta}{\Gamma, \forall\alpha.C \vdash \Delta} \qquad \frac{\Gamma \vdash C}{\Gamma \vdash \forall\alpha.C}$$

$$\textbf{Logic } (\exists)^1\textbf{:}\quad \frac{\Gamma, C \vdash \Delta}{\Gamma, \exists\alpha.C \vdash \Delta} \qquad \frac{\Gamma \vdash C[\tau/\alpha]}{\Gamma \vdash \exists\alpha.C}$$

We can now state a variation of the soundness theorem for this fragment of LJ: if $\Theta \vdash C_i$ for all i and $\Theta \vdash D$ are well-sorted constraint, and $\overrightarrow{C_i} \vdash D$ holds in LJ, then for any model $\Theta \models \sigma$ such that $\sigma \models C_i$ for all i, we have $\sigma \models D$. In other words, formal reasoning held in LJ preserve validity in all models.

Utility in the type system Since LJ and the focused linear logic we use for our type system are two instances of the same formalism, there is a natural way of combining them allowing logical reasoning on programs at type-level. This allows us to use parameters to denote sizes, lengths, amounts, etc. in a well-scoped manner compatible with first-order reasoning.

6.2 Integers and polynomials in constraints

The first-order signatures used in the full type-system can be specified by users, by declaring sorts, operations, constants, and relations within preludes. We introduce the sorts of *natural integers*, *resources*, and *polynomials*, which are required for our analysis scheme.

[1]Here, α must be renamed in C as to be fresh when the quantifier disappears as the rule is applied

Integers Natural integers have sort **nat**, and support addition and multiplication, with operators $(0, 1, +, *)$. Note that the partial operation of subtraction and divisions are not required. We can nevertheless describe the difference between two natural integers: instead of writing $C(n - m)$, we instead write $\exists k.(m + k = n) \wedge C(k)$, and likewise for division. This guarantees that models are always well-defined. Resources are, in practice, integers representing discrete amount of time, memory, or energy. As such, we give them sort **nat**, but also use the synonym **res** to clarify which parameters are resources.

Closed forms of shapes and resources Polynomials are the most important primitive we use at constraint-level. Indeed, the goal of the AARA is to infer a closed firm for a resource footprint. But our parameter system returns a constraint, which then needs to be solved to provide such closed form. In this work, those closed forms are multi-variate polynomials from $\mathbb{N}$ to $\mathbb{N}$. Other choices of formulas are of course possible, which would be achieved by a different specification of the first-order signature (for example, by adding logarithms or exponentials). Nevertheless, multi-variate polynomials have a considerable advantage as opposed to larger classes of closed forms: the first-order theory of polynomials with rational coefficients admits *quantifier elimination* [58, theorem 3.1.4]. This means the constraint generated by the type-system can be reduced to one in propositional logic, and then further into an optimization problem over the rationals, which can be solved by third party solvers to derive a minimal (or close to minimal) closed form of the resource footprint. We shall go into detail into this in chapter 7, but for now, we only need to specify the first-order sort we use for multi-variate polynomials, and introduce some useful syntactic sugar.

Multivariate polynomials Multivariate polynomials have a first order signature which encodes their algebraic structure and their support for composition. This is not obvious: indeed, given two polynomials $p, q : \mathbb{N}^2 \to \mathbb{N}$, defining their composition $r = \lambda x, y.p(q(x, y), y)$ apparently requires a binder to introduce variables. But it is also possible to use a first-order signature with the same effect.

Namely, multi-variate polynomials with coefficient in $\mathbb{N}$, domain $\mathbb{N}^k$, and image in $\mathbb{N}$ (called *polynomials* hereafter), are defined by the following first-order signature:

- Sorts $\mathbf{poly}_n$ for all $n \in \mathbb{N}$ for polynomials in n variables;

- Addition $+ : \mathbf{poly}_n \times \mathbf{poly}_n \to \mathbf{poly}_n$;
- Multiplication $\times : \mathbf{poly}_n \times \mathbf{poly}_k \to \mathbf{poly}_{n+k}$;
- Constant $cst : \mathbf{nat} \to \mathbf{poly}_0$;
- Variables $x_k^n : \mathbf{poly}_n$ for all $1 \leq k \leq n$ which represent $\lambda(x_1, \ldots, x_n).x_k$;
- Compositions $\circ : (\mathbf{poly}_n)^k \times \mathbf{poly}_k \to \mathbf{poly}_n$ defined by

$$(p_1, \ldots, p_k, q) \mapsto (p_1, \ldots, p_k) \circ q = \lambda \vec{x}.q(p_1(\vec{x}), \ldots, p_k(\vec{x}));$$

- Evaluation $eval : \mathbf{poly}_0 \to \mathbf{nat}$;
- Equality $p = q$ for $p : \mathbf{poly}_n$ and $q : \mathbf{poly}_k$;
- and comparaisons $p \leq q$ and $p < q$ for $p : \mathbf{poly}_n$ and $q : \mathbf{poly}_k$, defined by:

$$p \leq q \Leftrightarrow p(n) \leq q(n) \text{ asymptotically as } n \to \infty.$$

With this signature, algebraic expressions on naturals can be represented as polynomials, and common operations are available, in point-free style. For example, the polynomial $r(x, y) = p(q(x, y), y)$ from above is represented as $(q, x_2^2) \circ p$, and if α, β are two natural parameters, then the value of the expression $2p(q(\alpha, 3) + 1, \beta)$ is represented by:

$$eval((((cst(\alpha), cst(3)) \circ q) + cst(1), cst(\beta)) \circ p \times cst(2))$$

We shall freely use the informal notation for polynomial in the rest of this work. Furthermore, when Θ is a parameter scope, we write p_Θ for a polynomial variable, whose arguments are the variables of Θ of sort **nat**.

Having presented constraints and parameters, we may move on with the parameter-and-constraint type system proper. We shall now introduce this. To begin, we show how the **with** and **where** clauses in type definitions introduce constraints at type-level.

6.3 Typing with constraints

The extended type system with constraint and parameter uses the constraint language we introduced in chapter 4, reproduced here for convenience.

$$E ::= \top \mid R(\overrightarrow{\tau}) \wedge E \mid \tau = \tau' \wedge E$$
$$C ::= \top \mid \bot \mid E \wedge C \mid E \Rightarrow C \mid \forall \overrightarrow{\alpha : \mathbf{s}}.\, C \mid \exists \overrightarrow{\alpha : \mathbf{s}}.\, C$$

The judgments for the extended type system are also recalled below. They have the following form, in which the parameters in Θ are in scope in C, Γ, and Δ:

$$\begin{array}{rll} \textbf{Commands} & c : & (\Theta \models C) \triangleright (\Gamma \vdash \Delta) \\ \textbf{Terms} & & (\Theta \models C) \triangleright \Gamma \vdash t : A \\ \textbf{Environments} & & (\Theta \models C) \triangleright \Gamma \mid e : A \vdash \Delta \end{array}$$

With this added expressive power, it is possible to create static analyzers by adding parameters and simple constraints to type definitions with the **with** and **where** clauses. But first, let us cover some baselines.

Extending rules for simple types First, the rules we introduced last chapter for the simple type system all hold in extended variants in the full one. Those variants accumulate the parameter scopes and constraints of the premises into the conclusion. Here are some examples. The case of **commands** and **variables** are:

$$\frac{(\Theta \models C) \triangleright \Gamma \vdash t : A\,; \qquad (\Theta \models C) \triangleright \Gamma'\,; e : A \vdash \Delta}{\langle t \parallel e \rangle : (\Theta \models C) \triangleright (\Gamma, \Gamma' \vdash \Delta)}$$

$$\frac{}{(\Theta \models \top) \triangleright x : A \vdash x : A} \qquad \frac{}{(\Theta \models \top) \triangleright \emptyset \mid a : A \vdash a : A}$$

Binders have typing rules with only one premise. This means the first-order fragment is preserved as-is when rules for binders are applied (except for pattern-matching ones). We give the rule for the $\mu x^{-}.c$ binder. The same transformation applies as well for the three other binders, as well as for closures, thunks, exponentials, fixpoints, and pattern-matching with only one clause.

$$\frac{c : (\Theta \models C) \triangleright (\Gamma, x^{-} : A^{-} \vdash \Delta)}{(\Theta \models C) \triangleright (\Gamma \mid \mu x^{-}.c : A^{-} \vdash \Delta)}$$

In the case of a constructor with many arguments like functions, or a pattern-matching with many clauses (like sums), the first-order constraint for all different branches are added together. Note that the constraints are not accumulated with a disjunction, but a conjunction: all constraints must hold simultaneously.

$$\frac{(\Theta \models C) \triangleright \Gamma \vdash V : A \qquad (\Theta \models C) \triangleright \Gamma' \mid S : B \vdash \Delta}{(\Theta \models C) \triangleright \Gamma, \Gamma' \mid V \cdot S : A \multimap B \vdash \Delta}$$

$$\frac{c_1 : (\Theta \models C) \triangleright (\Gamma_1, x_1 : A_1 \vdash \Delta) \qquad c_2 : (\Theta \models C) \triangleright (\Gamma_2, x_2 : A_2 \vdash \Delta)}{(\Theta \models C) \triangleright \Gamma_1, \Gamma_2 \mid \mu(\iota_{1/2}(x_1).c_1 \mid \iota_{2/2}(x_2).c_2) : A_1 \oplus A_2 \vdash \Delta}$$

Extending the type system in this manner guarantees that every program typable in the simple type system as $(\Gamma \vdash \Delta)$ is typable in the extended type system as $(\emptyset \models \top) \triangleright (\Gamma \vdash \Delta)$. Note that if type definitions are modified to include parameters, this no longer true. Programs remain typable for *some* context Θ and constraint C.

Zero and Top In the simple type system, the types $\mathbb{0}$ and $\top$ types have "elimination" rules but no "introduction" rule. In the extended type system, we can introduce values of those types, provided the current constraint is unsatisfiable. This represents code that we know is dead because its related constraint cannot be satisfied. Formally, we add two new rules, allowing the creation of a value $\mathbb{0}_\Gamma$ and a value $\top_{\Gamma,\Delta}$. Those values incorporate references to the two current scopes as to preserve linearity. The two new rules are:

$$\frac{}{(\Theta \models \bot) \triangleright \Gamma \vdash \mathbb{0}_\Gamma : \mathbb{0}} \qquad \frac{}{(\Theta \models \bot) \triangleright \Gamma \mid \top_{\Gamma,\Delta} : \top \vdash \Delta}$$

Parameter and constraint management Some new rules in the full type system involve managing the parameter scope Θ and the current constraint C. First, we add weakening on parameters, which allow superfluous first-order variables to be removed from Θ (**fol-weak**). Then, we introduce a "subtyping" relation on constraints. This is to be understood as passing from a constraint with a large set of models to one with a smaller one, included in the first. (**fol-sub**).

$$\frac{(\Theta \models C) \triangleright (\Gamma \vdash \Delta) \qquad \mathrm{fv}(\Gamma, \Delta, C) \subset \Theta}{(\Theta, \Theta' \models C) \triangleright (\Gamma \vdash \Delta)}\ \textbf{(fol-weak)}$$

$$\frac{(\Theta \models C') \triangleright (\Gamma \vdash \Delta) \qquad \Theta \models C \Rightarrow C'}{(\Theta \models C) \triangleright (\Gamma \vdash \Delta)}\ \textbf{(fol-sub)}$$

Type annotation with parameters It is useful is practice to allow terms and environments to be annotated with types and parameters. To typecheck, it then suffices to unify the annotated and inferred type of the annotated expression. But unification of parameters is delegated to the constraint in our system. The rule for type annotations must therefore unload checking parameters for equality to the constraint. This leads to the following rule, where an annotated term $t : A(\vec{\tau})$ can be typed as $A(\vec{\tau'})$ (same monotype A, different parameters) when $\overrightarrow{\tau = \tau'}$ and $\vec{\tau}$ are well-scoped. The same holds for annotations on environments, values, stacks, and bound variables.

$$\frac{(\Theta \models C) \triangleright \Gamma \vdash t : A(\vec{\tau'}) \qquad \overrightarrow{\Theta \vdash \tau' : \mathbf{s}} \qquad \Theta \vdash A : \vec{\mathbf{s}} \to \pm}{(\Theta \models C \wedge \overrightarrow{\tau = \tau'}) \triangleright \Gamma \vdash (t^{\pm} : A(\vec{\tau})) : A(\tau)} \textbf{(term-annot)}$$

Equality in constraints Working with first-order equality is done with two rules which encode unification for first-order terms. Namely, **(fol-refl)** introduces a trivial equality $\alpha = \alpha$, and **(fol-unify)** allows a non-trivial equality $\tau = \tau'$ to be added to the current constraint when its two sides can be unified during type-checking. We show them for a generic typing sequent, but they apply to all expressions of System-**L**.

$$\frac{(\Theta \models C \wedge \alpha = \alpha) \triangleright (\Gamma \vdash \Delta)}{(\Theta \models C) \triangleright (\Gamma \vdash \Delta)} \textbf{(fol-refl)}$$

$$\frac{(\Theta \models C\theta) \triangleright (\Gamma\theta \vdash \Delta\theta) \qquad \theta = \mathrm{mgu}(\tau = \tau')}{(\Theta, x, y \models C \wedge \tau = \tau') \triangleright (\Gamma \vdash \Delta)} \textbf{(fol-unify)}$$

Those rule is often used in combination with **(fol-sub)** as to bring an equality $\tau = \tau'$ at the head of the constraint. Those two rules are not syntax-directed, but can be dispensed with in implementations. Indeed, substitution and equality of first-order terms can be handled with traditional unifiers.

6.4 Manipulating type-level constraints at term-level

We can now start showing the typing rules that manipulate constraints. Those will be introduced step-by-step. First, we introduce the existential quantifier, then its universal counterpart. Those two form a positive/negative pair. Then, we'll introduce a second pair, namely conjunction and implication. Once this is done, we will handle parameterized data and computation types, which combine the two pairs. Finally, those parameterized types will be used to defined analyzable data and computation.

Extending constraint scope with quantifiers Let us now begin in earnest. The first step is to extend the language with constructs for manipulating quantifiers. We want to do so in a way that matches both System-**L** and LJ. Recall the two rules dedicated to existential quantification in the LJ sequent calculus, where α is a fresh variable in the first rule.

$$\frac{\Gamma, C \vdash \Delta}{\Gamma, \exists\alpha.C \vdash \Delta} \qquad \frac{\Gamma \vdash C[\tau/\alpha], \Delta}{\Gamma \vdash \exists\alpha.C, \Delta}$$

Note furthermore, that the left rule for the existential quantifier is invertible in LJ. This tells us that the existential quantifier is a positive connector in LJ. As to match this, an existential quantifier should only be introduced by positive types in **L**. Likewise, universal quantifiers should only be introduced by negative types.

This justifies the following implementation. We define two **newtypes** with argument $A : \mathbf{s} \to \mathbf{b}$ that quantifies a parameter of sort **s**. In the positive case, the parameter is existential, and in the negative case it is universal. This parameter is bound by the **with** clause, whose purpose is to introduce new parameter variables.

newtype $\exists$ $(\mathsf{A}{:}\mathbf{s} \to$ **pos**$)$: **pos** = pack **of** $\mathsf{A}(\alpha)$ **with** $\alpha{:}\mathbf{s}$
newtype $\forall$ $(A : \mathbf{s} \to$ **neg**$)$: **neg** = spec **of** $\mathbb{1} \multimap A(\alpha)$ **with** $\alpha{:}\mathbf{s}$

The $\exists$ type defines a constructor pack which consumes a value $V : A(\tau)$ and introduces $\mathsf{pack}_\tau(V) : \exists(A)$, in which the parameter τ is hidden but witnessed to exist. On this other side, the binder $\mu\mathsf{pack}_\alpha(x).c$ binds V to x and τ to α. The inner command c doesn't know the value of τ, and must therefore be satisfied for all possible values. For this reason, the constraint for the binder is $\forall\alpha.C'$. This gives the following two rules:

$$\frac{(\Theta \models C[\tau/\alpha]) \triangleright \Gamma \vdash V : A(\tau)}{(\Theta \models \exists\alpha.C) \triangleright \Gamma \vdash \mathbf{pack}_\tau(V) : \exists(A)}\ (\exists\textbf{-R})$$

$$\frac{c : (\Theta, \alpha \models C) \triangleright (\Gamma, x : A \vdash \Delta)}{(\Theta \models \forall\alpha.C) \triangleright \Gamma \mid \mu\mathbf{pack}_\alpha(x).c : \exists(A) \vdash \Delta}\ (\exists\textbf{-L})$$

Universal quantifiers operate the same way: the stack constructor **spec** takes a stack of type $A(\tau)$ typed under $C[\alpha/\tau]$, and returns a stack of type $\forall(A)$ under constraint $\exists\alpha.C$. A universally quantified value introduces an arbitrary parameter α to stand for τ, and therefore is typed under some $\forall\alpha.C'$, which is what we wanted. This gives us the two rules below:

$$\frac{c : (\Theta, \alpha \models C) \triangleright (\Gamma \vdash a : A(\alpha))}{(\Theta \models \forall\alpha.C) \triangleright \Gamma \vdash \mu\mathbf{spec}_\alpha(a).c : \forall(A)}\ (\forall\textbf{-R})$$

$$\frac{(\Theta \models C[\tau/\alpha]) \triangleright \Gamma \mid S : A(\tau) \vdash \Delta}{(\Theta \models \exists\alpha.C) \triangleright \Gamma \mid \mathbf{spec}_\tau \cdot S : \forall(A) \vdash \Delta}\ (\forall\textbf{-L})$$

During reduction, the witness packaged with a constructor is substituted for the variable on the other side, leading to the following rules:

$$\langle \exists_\tau(\vec{V}) \parallel \mu\exists_\alpha(\vec{x}).c \rangle \triangleright c[\tau/\alpha, V/x] \qquad \textbf{(data)}$$

$$\langle \mu\forall_\alpha(a).c \parallel \forall_\tau \cdot S \rangle \triangleright c[\tau/\alpha, S/a] \qquad \textbf{(comput)}$$

Hypothesis management Following the same principle, we can add term-level additions to the language which handle the $E \wedge C$ and $E \Rightarrow C$ constraints. We shall introduce those two more briefly, as the same principle apply as for the last paragraph.

The constructor $A \wedge B$ is positive in LJ, as LJ admits the equivalence sequent $(\Gamma, E \wedge A \vdash \Delta) \leftrightarrow (\Gamma, E, A \vdash \Delta)$. Likewise, the $\Rightarrow$ constructor is negative, as $(\Gamma \vdash E \Rightarrow A, \Delta) \leftrightarrow (\Gamma, E \vdash A, \Delta)$. This justifies the interpretation of the **where** E clauses, which introduce simple constraints. By analogy with dependent type theory, we define, for any simple constraint $\overrightarrow{\alpha : \mathbf{s}} \vdash E$ two types as:

newtype $E \wedge (-)$ $(\mathsf{A} : \vec{\mathbf{s}} \to \mathbf{pos})(\overrightarrow{\alpha : \mathbf{s}})$: **pos** = witness **of** $\mathsf{A}(\vec{\alpha})$ **where** E
newtype $E \Rightarrow (-)$ $(\mathsf{A} : \vec{\mathbf{s}} \to \mathbf{neg})(\overrightarrow{\alpha : \mathbf{s}})$: **neg** = assume **of** $\mathsf{A}(\vec{\alpha})$ **where** E

For the $E \wedge A$ type, the right rule adds E to the constraint, and the left rule (the eliminator) assumes E, which has already been witnessed on the other side. This is reverse the $E \Rightarrow A$: the right side assumes E, and the left side must witness it. The generated typing rules are:

$$\frac{(\Theta \models C) \triangleright \Gamma \vdash V : A(\vec{\tau})}{(\Theta \models E[\overrightarrow{\tau/\alpha}] \wedge C) \triangleright \Gamma \vdash \mathbf{witness}_E(V) : (E \wedge A)(\vec{\tau})}\ (\wedge\textbf{-R})$$

$$\frac{c : (\Theta \models C) \triangleright (\Gamma, x : A(\vec{\tau}) \vdash \Delta)}{(\Theta \models E[\overrightarrow{\tau/\alpha}] \Rightarrow C) \triangleright \Gamma \mid \mu\mathbf{witness}_E(x).c : (E \wedge A)(\vec{\tau}) \vdash \Delta}\ (\wedge\textbf{-L})$$

$$\frac{c : (\Theta \models C) \triangleright (\Gamma \vdash a : A(\vec{\tau}))}{(\Theta \models E[\overrightarrow{\tau/\alpha}] \Rightarrow C) \triangleright \Gamma \vdash \mu\mathbf{assume}_E(a).c : (E \Rightarrow A)(\vec{\tau})}\ (\Rightarrow\textbf{-R})$$

$$\frac{(\Theta \models C) \triangleright \Gamma \mid S : A(\vec{\tau}) \vdash \Delta}{(\Theta \models E[\overrightarrow{\tau/\alpha}] \Rightarrow C) \triangleright \Gamma \mid \mathbf{assume} \cdot S : (E \Rightarrow A)(\vec{\tau}) \vdash \Delta}\ (\Rightarrow\textbf{-L})$$

This concludes the addition of constraint management to the type system. In summary, the **with** and **where** clauses of newtypes allows simply-typed programs to be enriched for constraint-level manipulations. As to simplify the rest of the presentation, we freely write the new types as $\exists\alpha.A$, $E \wedge A$, etc. This principle is applied for datatypes and computation types, on a construct-per-constructor basis, which let us define such types as nested data structures with shape-aware types, and resource pools.

6.5 Defining parameterized types

We now combine data and computation types with the **where** and **with** clauses, which allows us to define resource aware structures in the machine, but also to relate the shape of a value and the resources assigned to it. We begin by giving an example of this added power in the context of list data structures. We then move on to typing parameterized datatypes, and finally parameterized computation types.

Parameterized data structures Typing parameter-aware data structures requires changing the argument's types. For example, using lists defined as an ADT $\mathrm{List}(A)$, a list of lists of integer may be represented as $\mathrm{List}(\mathrm{List}(\mathbb{N}))$. When adding length parameters into play, there is only so much information once can encode at type level while keeping List a type constructor of sort $\mathbf{pos} \to \mathbf{pos}$.

Consider instead a refined list constructor using a **nat** integer parameter to encode length as $\mathrm{List} : (\mathbf{nat}, \mathbf{pos}) \to \mathbf{pos}$. This allows bounding the types such as:

- $\exists\alpha, \beta.\, \mathrm{List}(\alpha, \mathrm{List}(\beta, \mathbb{N}))$, lists of α lists of β integer;
- $\exists\alpha.\, \mathrm{List}(\alpha, \mathrm{List}(\alpha, \mathbb{N}))$, matrix-like values of $\alpha \times \alpha$ integer entries;
- $\exists\alpha.\, (\alpha < K) \wedge \mathrm{List}(\alpha, \exists\beta.(\beta < L) \wedge \mathrm{List}(\beta, \mathbb{N}))$; another matrix-like list of list, whose shape is contained in a $K \times L$ rectangle. This type is equivalent to
$$\exists\alpha, \beta.\, (\alpha \leq K \wedge \beta \leq L) \wedge \mathrm{List}(\alpha, \mathrm{List}(\beta, \mathbb{N}))$$
- More generally, any rectangular shape which can be defined by a first-order equation when other parameters are in scope, by replacing $(\alpha \leq K \wedge \beta < L)$ by an arbitrary system of (in)equations $E[\alpha, \beta]$.

This offers some flexibility, but doesn't allow for a significant kind of dependency in shape: The length of an inner lists cannot depend on its position within the outer one. This means that, for example, triangular shapes, where the i^{th} element has length i cannot be encoded. But since type constructors can have arguments depending on parameters, we can encode more shapes of nested data structures. Consider new a type constructor for linked lists defined as

$$\text{List} : (\mathbf{nat} \to \mathbf{pos}, \mathbf{nat}) \to \mathbf{pos}$$

A list of type $\text{List}(\lambda\alpha.A(\alpha), \beta)$ is a list of β elements, where the ith element has type $A(i)$. For example, the type $\text{List}(\lambda\alpha.\text{List}(\mathbb{N}, \alpha + 1), \beta)$ is the triangle-shaped list of list we couldn't define before. For another example, let us say we want the length of the inner lists to *decrease* as the index in the outer list increases. This could be realized by passing the parameter $\beta - \alpha$ to A, but subtraction isn't total on $\mathbb{N}$, which pauses technical issues. On the other end, we can define the type:

$$\exists\beta.\, \text{List}(\lambda\alpha.\, \exists\gamma.\, (\gamma + \alpha = \beta) \wedge \text{List}(\mathbb{N}, \gamma), \beta)$$

This has the triangular, decreasing shape we seek, using a constraint and the well-defined addition operator on natural numbers: the first inner list has type $\text{List}(\mathbb{N}, \beta)$, the second has type $\text{List}(\mathbb{N}, \beta - 1)$, etc. Using parameterized type constructors and first-order constraints together, the shape of data structures can be successfully tracked at compile time.

Parameterized datatypes Recall that datatypes are defined with the following syntax, which involves parameters arguments and bound parameters per-constructor in **where** clauses, both of them subject to a constraint in **with** clauses:

$$\begin{array}{l}
\textbf{data}\ \ K^{+}(\vec{\mathsf{A}}{:}\vec{\mathbf{m}})\ \ (\vec{\alpha}{:}\vec{\mathbf{s}})\ \ = \\
\quad |\ \ k_1\ \ \textbf{of}\ \ \overrightarrow{\mathsf{B}_1^{+}}\ \ \textbf{where}\ \overrightarrow{\beta_1}{:}\overrightarrow{\mathbf{s}'_1}\ \textbf{with}\ E_1 \\
\quad \ldots \\
\quad |\ \ k_m\ \ \textbf{of}\ \ \overrightarrow{\mathsf{B}_m^{+}}\ \ \textbf{where}\ \overrightarrow{\beta_m}{:}\overrightarrow{\mathbf{s}'_m}\ \textbf{with}\ E_m \\
\textbf{end}
\end{array}$$

In this definition, The type constructor K is a datatype with monotype arguments $\vec{A}$ and parameter arguments $\vec{\alpha}$. Each constructor k_i has type-level parameter arguments $\vec{\beta_i}$, which

must satisfy E_i. Each constructor defines a judgment defining its acceptable instantiations, written as:

$$k_i : (\overrightarrow{\beta_i}, E_i, \overrightarrow{B_i}) \to K(\overrightarrow{A})(\overrightarrow{\alpha})$$

For example, lists of length α and elements of types $A(0)$, $A(1)$, etc. are definable as:

```
data List(A: nat → pos) (α:nat) =
  | cons of A(α) ⊗ List(A,β) where β:nat with (α = β+1)
  | nil with (α = 0)
end
```

For parameterized lists, the values for α and each β of each Cons are completely determined: the parameters are only used for *computation* not *specification*. But we also allow partial specifications of parameters, in which case the parameters are *reasoned on*. For example, consider two definitions of the Nat type using Church encoding. The first one provides integers typed as $\mathsf{Nat}(\alpha)$, where α : **nat** is a natural parameter specifying the exact value of the integer.

```
data Nat (α:nat) =
  | z with α = 0
  | s of Nat(α′) with α′ with α = α′ + 1
end
```

The second one shown below, on the other hand, has two parameters α, β : **nat**, specifying an upper and lower bound for the value of that integer. This means that, at type-level, it is known that the integer belongs to the interval $[\alpha, \beta]$, but no more information is available. It is implemented by stating that the value 0 belongs to every type-level interval $[0, \beta]$, and that $n + 1$ belongs to $[\alpha, \beta]$ whenever n belongs to $[\alpha', \beta']$ with $\alpha \leq \alpha' + 1$ and $\beta' + 1 \leq \beta$.

```
data Nat (α:nat, β:nat) =
  | z with α = 0
  | s of Nat(α′,β′) with α′,β′:nat with α ≤ α′ + 1 ∧ β′ + 1 ≤ β
end
```

Parameterized computation types Recall, like before, the definition of a computation type:

```
comput K⁻(A⃗:m⃗) (α⃗:s⃗) =
  | k_1 of B_1^+⃗ ⊸ B'_1⁻ where β_1⃗:s'_1⃗ with E_1
  ...
  | k_m of B_m^+⃗ ⊸ B'_m⁻ where β_m⃗:s'_m⃗ with E_m
end
```

Using those, we can define, for example, the parameterized type of streams, which lazily provides values of type $A(0)$, $A(1)$, etc., and canbe destroyed to recover an internal state of type $B(\alpha)$, where α denotes the number of elements already emitted.

```
comput Stream(A: nat → pos, B : nat → pos) (α:nat) =
  | next of 𝟙 ⊸ ⇑(A(α) ⊗ ⇓Stream(β)) where β:nat with β = α+1
  | destroy of 𝟙 ⊸ ⇑B(α)
end
```

A resource token for safe (de-)allocation Parameterized types can be used to define static approximations of program footprints. Recall that with the physicist's method for amortized complexity, a program holds two kinds of resources, *used* and *free*, whose sum remains constant as programs reduce. This can be encoded as a parameterized datatype.

To this end, consider a free parameter variable T : **nat** representing the total footprint. At a given point of evaluation, we can represent the state of resources as a pair (F, U) of two **nat** parameters (U for used and F for free) constrained by the equation $F + U = T$. Without loss of generality, the program can be considered to begin with no allocated resources, such that $F = T$ and $U = 0$. During execution, resources can be allocated, causing a transition from $(F + K, U)$ to $(F, K + U)$ keeping T constant. The converse happens when resources are freed. We can encode this behavior with the parameterized datatype $ST(F, U)$ defined as:

```
data S(F,U: nat) =
  | alloc of ST(F',U') with K:nat where U'=U+K ∧ F=F'+K
  | free  of ST(F',U') with K:nat where U=U'+K ∧ F'=F+K
end
```

We often abbreviate $\mathrm{S}(F, U)$ as S_U^F. With this definition, allocation of K resources is just packing a token st into $\mathsf{alloc}_K(st)$, and likewise for freeing. Note that no closed value of type $\mathrm{S}(F, U)$, and that there is no way to destroy a token, since it is a linear type. This means that

a program wishing to manipulate resources must take a token as argument and return it as part of its final result.

Also, note that, as resource manipulations accumulate and constructors are layered on top of the token, they create a log of all resource manipulations done in the past. This preservation the resource history during evaluation is crucial. Indeed, of the case of a program that uses a lot of resources in a first half of its execution, but less in the second half, the footprint required for the second half will be lower than the total footprint over the entire execution: the footprint of a program is not preserved by reduction. On the other hand, with our setup, the history of this higher resource requirement is preserved, and therefore the higher footprint is as well. This type shall be the keystone of our analysis, and shall be used extensively in chapter 6.

Having shown how parameterized types are defined, let us now show how typing rules deal with the token and another parameterized types defined by constructors.

6.6 Using parameterized types

The typing rules for parameterized datatypes are merely a mix of those for simple datatypes, existential quantification, and witnessing. When typing the application of a constructor $k_{\vec{\tau'}}(\vec{V})$, the constructor is first instantiated as some $k : \exists\vec{\alpha}.E \wedge \vec{U} \to K(\vec{A})(\vec{\tau})$ from its definition. Then, the arguments $\vec{V}$ are typed against $\vec{U}$, and the witnesses $\vec{\tau'}$ are unified with $\vec{\alpha}$, which yields a constraint $\Theta \models E[\overrightarrow{\tau'/\alpha}]$. The constructor application is then typed as $K(\vec{A})(\vec{\tau})$. This is rule **(data-R)**.

$$\frac{\overrightarrow{(\Theta, \vec{\alpha} \models C) \triangleright \Gamma \vdash V : U} \qquad \vdash k : \exists\vec{\alpha}.E \wedge \vec{U} \to K(\vec{A})(\vec{\tau})}{(\vec{\Theta} \models \exists\vec{\alpha}.\, C \wedge E \wedge \overrightarrow{\alpha = \tau'}) \triangleright\ \vec{\Gamma} \vdash k_{\vec{\tau'}}(\vec{V}) : K(\vec{A})(\vec{\tau})}\ \textbf{(data-R)}$$

When typing a pattern matching clause $k_{\vec{\alpha}}(\vec{x}).c$ with the same constructor, a fresh instance of the constructor is likewise generated. The parameters α and variables $\overrightarrow{x : U}$ are added to the scope, and the command c is typed, giving a constraint C. Then, the $\vec{\alpha}$ are quantified universally and their constraint E is assumed to hold. The value the clause matches against is meant will provide terms for the α and a witness of E. The overall constraint is then $\forall\alpha.\, E \Rightarrow C$. This is rule **(clause-L)**. When many clauses are present, they are checked for exhaustivity and their respective constraints are accumulated. This is rule **(data-L)**.

$$\frac{\vdash k : \exists.\vec{\alpha}.E \wedge \vec{U} \to K(\vec{A})(\vec{\tau}) \qquad c : (\Theta, \vec{\alpha} \models C) \triangleright (\Gamma, \overrightarrow{x : U} \vdash \Delta)}{(\Theta \models \forall\vec{\alpha}.\, E \Rightarrow C) \triangleright \Gamma \mid k_\alpha(\vec{x}).c : K(\vec{A})(\vec{\tau}) \vdash \Delta \ \mathbf{cl}.}\ \textbf{(clause-L)}$$

$$\frac{\vdash \vec{k^i} \twoheadrightarrow K \qquad \overrightarrow{(\Theta \models C) \triangleright \Gamma \mid k_\alpha(\vec{x}).c : K(\vec{A})(\vec{\tau}) \vdash \Delta \ \mathbf{cl}.}}{(\vec{\Theta} \models \vec{C} \triangleright \Gamma \mid \mu(\overrightarrow{k_{\vec{\alpha}}(\vec{x}).c}) : K(\vec{A})(\vec{\tau}) \vdash \Delta}\ \textbf{(data-L)}$$

Parameterized computation types and parameterized newtypes follow the same principle, where constructor application introduces $\exists$ and $\wedge$ at type level, and clauses introduce a matching $\forall$ and $\Rightarrow$.

6.7 Closing remarks: A resource-aware program

In this chapter, we have extended the simple type system of System-**L**, namely intuitionistic linear logic, with a first-order fragment which allows types to be parameterized by first-order variables. Those first-order variables are then subject to a constraint, and typing holds only for the values of those parameters which satisfy this constraint.

The language of types was extended with the capability to introduce existential and universal quantification, and witnessing and assumption of first-order relations between parameters. Those types are transparent at runtime: then can be erased before execution. Furthermore, those reasoning primitives were embedded within algebraic datatypes and computation types. This enabled the shape of nested data structures, the usage of computations, and the ressource footprint of programs to be approximated at compile-time.

To close this chapter, we give an example of a time analysis of the *append* function on linked lists. For reference, an OCaml implementation of those functions is:

```
let rec rev_append l1 l2 = match l1 with [] -> l2
    | h::t -> rev_append t (h::l2)
let append l1 l2 = rev_append (rev_append l1 []) l2
```

Below is the code of those two functions, together with their simple types. We write $(-)$ for the erased parameters, and S for the resource token type.

$$
\begin{aligned}
&\text{revappend} : \ !\,\mathbf{fix}\,\forall(-).\ S \multimap L(A) \multimap L(A) \multimap \Uparrow(S \otimes L(A)) = \\
&\quad \mu(\mathbf{fix} \cdot a).\langle \mathbf{self} \parallel \mu f. \\
&\quad\ \langle \mu(\mathbf{spec}_{(-)} \cdot st \cdot l_1 \cdot l_2 \cdot b). \\
&\quad\quad \langle l_1 \parallel \mu \\
&\quad\quad\quad \mid \mathrm{nil}.\langle \mu(\Uparrow d.\langle st \otimes l_2 \parallel d\rangle \parallel b\rangle \\
&\quad\quad\quad \mid \mathrm{cons}_{n'}(h,t).\langle f \parallel\ !\cdot \mathbf{fix} \cdot \mathbf{spec}_{(-)} \cdot \mathrm{alloc}_1(st) \cdot t \cdot \mathrm{cons}_{(-)}(h, l_2) \cdot b\rangle\rangle \\
&\quad\ \parallel a\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\text{append} : \ \forall(-).\ S \multimap L(A) \multimap L(A) \multimap \Uparrow(S \otimes L(A)) = \\
&\quad \mu(\mathbf{spec}_{(-)} \cdot st \cdot l_1 \cdot l_2 \cdot \Uparrow \cdot a). \\
&\quad\ \langle \text{revappend} \parallel\ !\cdot \mathbf{fix} \cdot \mathbf{spec}_{(-)} \cdot st \cdot l_1 \cdot \mathrm{nil} \cdot \Uparrow \cdot \mu(st' \otimes l_1'). \\
&\quad\quad \langle \text{revappend} \parallel\ !\cdot \mathbf{fix} \cdot \mathbf{spec}_{(-)} \cdot st' \cdot l_1' \cdot l_2 \cdot \Uparrow \cdot a\rangle\rangle
\end{aligned}
$$

When typing this function with the full type system, those two functions can both be typed under the $\top$ constraint, and given the following types.

$$
\begin{aligned}
&\text{revappend} : \ !\,\mathbf{fix}\,\forall n, m, F, U.\ \mathrm{S}_U^{n+m+F} \multimap L(A, n) \multimap L(A, m) \\
&\qquad \multimap \Uparrow \mathrm{S}^F_{n+m+U} \otimes L(A, n+m)
\end{aligned}
$$

$$
\begin{aligned}
&\text{append} : \ \forall n, m, F, U.\ \mathrm{S}_U^{F+2n+m} \multimap L(A, n) \multimap L(A, m) \\
&\qquad \multimap \Uparrow \mathrm{S}^F_{2n+m+U} \otimes L(A, n+m)
\end{aligned}
$$

Here, n and m denote the lengths of the two lists to be appended, U and F denote the used and free resources, and the resource token type is S^F_U. Both *append* and *revappend* pass the token as state, and a unit cost is used each time a new *cons* constructor is used by *revappend*.

As we can see, a call to *revappend*, which reverse-and-appends a list of length n to the left of a list of length m uses n resources, and a call to *append* uses $2n$ resources, and is therefore independent of the length of the second list. This is a basic example of the kind of analysis we can automatically undertake thanks to our time system.

We can now move on to porting this analysis to high-level programming languages in the ML family, and on to extending it to support more sophisticated source programming and get finer bounds. This is the purpose of the next chapter.

6.8 Soundness of constraint generation

We now relate our constraint annotations to the runtime behavior of the System-**L**. The main point is that, when a command generates a constraint, models of that constraint remain valid as the command is evaluated. We do not provide full proofs of those results here, merely sketches of the salient points. The reader can safely skip this section on a first reading.

First state a lemma regarding constraints. Consider a constraint C using the literal $\top$. We write $C = C[\top]$ implicitly using the notion of constraint context. Note that $\top$ cannot occur as part of an equation E with this definition. By induction on the syntax of constraints, if $\Theta \vdash C[\top] \wedge C'$, then $\Theta \vdash C[C']$. That is, moving a constraint C' deeper within another C preserve provability, and therefore models.

We can then give a first result, which is that value substitution preserves models. Stack substitution proceeds the same way. Consider a typed command $c : (\Theta \models C) \triangleright (\Gamma, x : A \vdash \Delta)$ and a typed value $(\Theta' \models C') \triangleright \Gamma' \vdash V : A$. The instance of x in c generates a constraint $\top$ within C, which gives the following typing of $\langle V \parallel \mu x.c \rangle$:

$$\dfrac{\dfrac{\vdots}{(\Theta' \models C') \triangleright \Gamma' \vdash V : A} \qquad \dfrac{\dfrac{\dfrac{}{(\Theta'' \models \top) \triangleright x : A \vdash x : A}}{\vdots}}{\dfrac{c : (\Theta \models C[\top]) \triangleright (\Gamma, x : A \vdash \Delta)}{(\Theta \models C[\top]) \triangleright \Gamma \mid \mu x.c : A \vdash \Delta}}}{\langle V \parallel \mu x.c \rangle : (\Theta, \Theta' \models C[\top] \wedge C') \triangleright (\Gamma, \Gamma' \vdash \Delta)}$$

Then, $\langle V \parallel \mu x.c \rangle$ reduces to $c[V/x]$, which has the following typing, where the typing of V has replaced the one for x:

$$\dfrac{\dfrac{}{(\Theta' \models C') \triangleright \Gamma' \vdash V : A}}{\vdots}$$
$$c[V/x] : (\Theta, \Theta' \models C[C']) \triangleright (\Gamma, \Gamma' \vdash \Delta)$$

Therefore, the constraint $C' \wedge C[\top]$ of $\langle V \parallel \mu x.c \rangle$ implies the constraint $C[C']$ of $c[V/x]$. This forms the basis of our proof of soundness.

Indeed, this first result suffices to prove, by induction, that all other reduction rules, except those for parameterized data/computation types, also preserve models. In the case of the rule

(struct), which implement scope management, multi-hole contexts are required. Remains to show that reduction of parameterized types, rules **(data)** and **comput**, also preserve models.

We now show that reduction using the **(data)** rule preserves models. As a matter of legibility, we only present to the case where the constructor involved only has one argument. The case of many arguments, and of **comput** are similar. Let us consider a constructor with one argument, specified as $k : (\vec{\beta}, E(\vec{\beta}), B(\vec{\beta})) \to K(\vec{A})(\vec{\alpha})$. We write T for $K(\vec{A})(\vec{\alpha})$. By definition, a well-typed command that reduces with **(data)** using that constructor is typed in the following way, which we describe in three steps:

- A value $k(V)$ is typed. The inner value V has type $B(\vec{\tau})$ under constraint C_1.

$$\dfrac{\dfrac{\vdots}{(\Theta_1 \models C_1[\vec{\tau}/\vec{\beta}]) \triangleright \Gamma_1 \vdash V : B(\vec{\tau})}}{(\Theta_1 \models C_2) \triangleright \Gamma_1 \vdash k_{\vec{\tau}}(V) : T}$$

$$\begin{aligned} \textit{with: } & \Theta_1 \vdash \vec{\tau} = \dots \quad \textit{(witnesses for } V\textit{)} \\ & \Theta_1 \vdash C_1 = \dots \quad \textit{(constraint for } V\textit{)} \\ & \Theta_1 \vdash C_2 = \exists \beta_2.\, E[\beta_2] \wedge (\vec{\tau} = \vec{\beta}_2) \wedge C_1 \quad \textit{(value constraint)} \end{aligned}$$

- A pattern matching clause $k(x).c$ is typed. This involves creating free parameters $\vec{\beta_3}$ quantified universally, which are used when typing x. When c is typed, it generates a constraint $C_3(\beta_3) = C_3(\beta_3)[\top]$ which is a constraint context over the constraint $\top$ generated when typing x:

$$\dfrac{\dfrac{\dfrac{(\vec{\beta_3} \models \top) \triangleright x : B(\vec{\beta_3}) \vdash x : B(\vec{\beta_3})}{\vdots}}{c : (\Theta_3, \vec{\beta_3} \models C_3(\vec{\beta_3})[\top]) \triangleright (\Gamma_3, x : B(\vec{\beta_3}) \vdash \Delta)}}{(\Theta_3 \models C_4) \triangleright \Gamma_3 \mid k_{\vec{\beta_3}}(x).c : T \vdash \Delta \ \mathbf{cl}.}$$

$$\begin{aligned} \textit{with: } & \Theta_3, \beta_3 \vdash C_3(\vec{\beta_3})[\top] = \dots \quad \textit{(constraint context for } c\textit{)} \\ & \Theta_3 \vdash C_4 = \forall \beta_3.\, E[\beta_3] \Rightarrow C_3(\vec{\beta_3})[\top] \quad \textit{(stack constraint)} \end{aligned}$$

- Finally, the value and clause are combined in a command. The stack of this command also involves other clauses, which will not match the value. We write Θ_5 their collective

first-order scope, and C_5 their collective first-order constraint.

$$\frac{\dfrac{\vdots}{(\Theta_1 \models C_2) \triangleright \Gamma_1 \vdash k(V) : A} \quad \dfrac{\vdots}{(\Theta_3 \models C_4) \triangleright \Gamma_3 \mid k(x).c : A \vdash \Delta \ \mathbf{cl}.} \quad \ldots}{\langle k_{\vec{\tau}}(V) \parallel \mu(k_{\overrightarrow{\beta_3}}(x).c \mid \ldots)\rangle : (\Theta_1, \Theta_3, \Theta_5 \models C_2 \wedge C_4 \wedge C_5) \triangleright (\Gamma_1, \Gamma_4 \vdash \Delta)}$$

$$\textit{with: } \Theta_5 = \ldots \quad \textit{(scope for the non-matching clauses)}$$
$$\Theta_5 \vdash C_5 = \ldots \quad \textit{(constraint for the non-matching clauses)}$$

This yields, for the overall command, the constraint $\Theta_1, \Theta_3, \Theta_5 \models C_2 \wedge C_4 \wedge C_5$, which we now have to compare with the command after reduction. The reduction in question is $\langle k_{\vec{\tau}}(V) \parallel \mu(k_{\overrightarrow{\beta_3}}(x).c \mid \ldots)\rangle \triangleright c[\vec{\tau}/\overrightarrow{\beta}, V/x]$, which types as:

$$\frac{\dfrac{\dfrac{\vdots}{(\Theta_1 \models C_1) \triangleright \Gamma_1 \vdash V : B(\tau)}}{\vdots}}{c[\vec{\tau}/\overrightarrow{\beta_3}, V/x] : (\Theta_1, \Theta_3 \models C_3(\vec{\tau})[C_1]) \triangleright (\Gamma_1, \Gamma_3 \vdash \Delta)}$$

We therefore have to prove the following statement:

For:

$\Theta_1, \Theta_3, \Theta_5$ arbitrary contexts

$\Theta_1 \vdash C_1$ an arbitrary constraint

$\Theta_3, \overrightarrow{\beta_3} \vdash C_3(\overrightarrow{\beta_3})[]$ an arbitrary context

$\Theta_5 \vdash C_5$ an arbitrary constraint

$\Theta_1 \vdash C_2 = \ \exists \overrightarrow{\beta_2}.\, E[\overrightarrow{\beta_2}] \wedge (\vec{\tau} = \overrightarrow{\beta_2}) \wedge C_1$

$\Theta_3 \vdash C_4 = \ \forall \overrightarrow{\beta_3}.\, E[\overrightarrow{\beta_3}] \Rightarrow C_3(\overrightarrow{\beta_3})[\top]$

If :

$\Theta_1, \Theta_3, \Theta_5 \models C_2 \wedge C_4 \wedge C_5$

Then:

$\Theta_1, \Theta_3 \models C_3(\vec{\tau})[C_1]$

If suffices to show that the provability of the first implies the provability of the second. We do so with the following chains of formal reasoning steps, which concludes the proof.

$\Theta_1, \Theta_3, \Theta_5 \vdash C_2 \wedge C_4 \wedge C_5$

$\Theta_1, \Theta_3 \vdash C_2 \wedge C_4$ *by elimination for* $\wedge$

$\Theta_1, \Theta_3 \vdash (\exists \overrightarrow{\beta_2}.\, E[\overrightarrow{\beta_2}] \wedge C_1 \wedge (\overrightarrow{\tau} = \overrightarrow{\beta_2})) \wedge C_4$ *by definition of* C_2

$\Theta_1, \Theta_3, \overrightarrow{\beta_2} \vdash E[\overrightarrow{\beta_2}] \wedge C_1 \wedge (\overrightarrow{\tau} = \overrightarrow{\beta_2}) \wedge C_4$ *by elimination of* $\exists$

$\Theta_1, \Theta_3 \vdash E[\overrightarrow{\tau}] \wedge C_1 \wedge C_4$ *by substitution of* $\overrightarrow{\beta_2} = \overrightarrow{\tau}$

$\Theta_1, \Theta_3 \vdash E[\overrightarrow{\tau}] \wedge C_1 \wedge \forall \overrightarrow{\beta_3}.\, E[\overrightarrow{\beta_3}] \Rightarrow C_3(\overrightarrow{\beta_3})[\top]$ *by definition of* C_4

$\Theta_1, \Theta_3 \vdash E[\overrightarrow{\tau}] \wedge C_1 \wedge E[\overrightarrow{\tau}] \Rightarrow C_3(\overrightarrow{\tau})[\top]$ *by elimination of* $\forall$

$\Theta_1, \Theta_3 \vdash C_1 \wedge C_3(\overrightarrow{\tau})[\top]$ *by elimination of* $\Rightarrow$

$\Theta_1, \Theta_3 \vdash C_3(\overrightarrow{\tau})[C_1]$ *by the previous lemma*

Chapter 7

Implementing resource analysis for System-**L**

At the end of the previous chapter, we presented the code of an implementation of the *append* function in System-**L**. Its type verified the time complexity of this implementation. The question is, how can this setup be systematized, to a large corpus of programs and arbitrary (step-wise statically computable) cost metrics ?

This is the topic of this chapter. We show how to automatically transform programs to-and-from the simply-typed System-**L** as to explicitly encode resource manipulation at type-level. This is done by enriching programs with resource-passing, cost centers, and potential on shared values. This produces programs that implement type-level resource analyses. We use this as a backend to an analysis framework for ML-style languages. Since our frontend embeds pure ML programs into the simply-typed System-**L**, this next step suffices to create resource-aware versions of source programs. After this transformation pass, a type-inference phase in the parameterized type system will generate a constraint representing the resource footprint of the program under analysis.

Taken as a whole, those fully-automated transformations turn programs and type definitions from simply-typed ML into shape/complexity/resource-aware ones that implemented a generic resource analysis. The parameter constraints of the transformed programs generate specify their resource footprints, while changes to type definitions encode which parameters can footprints depend on, and the models of potential metric chosen by solvers determine a space of potential functions under consideration.

To begin with, we present the general principle of the transformation and its main primitives in section 7.1. Then, we present the transformation itself in two parts. This is done with a *CBPV effect* that automatically passes around a central *resource token* through programs. We present its linear fragment in section 7.2 and its non-linear fragment in section 7.3. We then present the implementation of the token and of potential primitives in section 7.4, which completes the presentation. Section 7.5 is dedicated to presenting a soundness result, and 7.6 to the automated translation of type definitions that implement polynomial AARA.

7.1 First steps

The input of the translation is a simply-typed System-**L** command, together with a parameterized prelude of type definitions. It outputs a parameterized resource-aware command ready for parameterized type inference. This transform uses some primitives we define later, as to factor the presentation.

Principle The transformation has two goals, which are achieved simultaneously: to pass a resource token along control flow in programs, and introduce binders for parameters as to quantify free parameters and constraints that will occur when type-checking. The example of CBV function is enlightening. A function $\Downarrow A \multimap \Uparrow B$ takes control when $\Downarrow$ is opened, consumes an A, then yields control when $\Uparrow$ is forced, and produces a B. Its transformed will be typed as:

$$\Downarrow \forall \vec{\alpha}.\ E \Rightarrow \mathrm{S}_U^F A \multimap \Uparrow \exists \vec{\beta}.\ E' \wedge \mathrm{S}_{U'}^{F'} B$$

This transformed function takes control when the closure is opened, takes the resource token[1] S_U^F and an argument A, for some $\vec{\alpha}$ such that E. Then, when the thunk is forced, it yields control, a new token $\mathrm{S}_{U'}^{F'}$ and a result B, parameterized by some $\vec{\beta}$ such that E'. The function works *for all* parameters $\vec{\alpha}$ such that E, *all* tokens S_U^F and *all* arguments A, and returns *some* parameters $\vec{\beta}$ such that E', *some* token $\mathrm{S}_{U'}^{F'}$ and *some* return value B.

This generalizes to arbitrary types: any closure takes control and accepts *all* calls with parameters A and ressources S_U^F as long their parameters α validate some constraint E'; thunks yield control over *some* data and resources, specified by parameters $\vec{\beta}$ such that E'.

[1] Namely, a *State token* $\mathrm{S}_U^F$with a *free* resource amount F and an *used* resource amount U, whereby the notation S_U^F.

This justifies the following translation scheme. We write the transformations to-and-from expressions and to-and-from types using double-brackets $[\![\ldots]\!]$. The translation of closures and thunks are:

$$\begin{aligned}[\![\Downarrow A^-]\!] &= \Downarrow \forall \vec{\alpha}.\, E \Rightarrow \mathrm{S}_U^F \multimap [\![A^-]\!] \\ [\![\Uparrow B^+]\!] &= \Uparrow \exists \vec{\beta}.\, E' \wedge \mathrm{S}_{U'}^{F'} \otimes [\![B^+]\!]\end{aligned}$$

The translation does just that, extending programs with token-passing through the thunks and closures, and inserting inference points that will eventually be filled out with quantifiers $\forall$ or $\exists$ and constraint $E \Rightarrow (-)$ or $E \wedge (-)$. Using this scheme as a backbone, costs and potentials are encoded when relevant using primitives which we describe in the next section.

Primitives The translation does not manipulate parameters, but instead uses some types in which placeholders are used to represent parameters and constraints to be instantiated in the future. Those are:

- The token type $S_?$, understood as the type S_U^F where the parameters F and U are yet unknown;
- A positive type $\mathrm{Pot}_?(A^+)$, which represents exponential values of type $!A^+$ charged with indeterminate amount of potential.
- A positive type $\exists_?(-)$ for the existential binder, which is to be filled out by a type $\exists \vec{\alpha}.\, E \wedge (-)$, together with its constructor $\exists_?(V)$;
- A negative type $\forall_?(A^-)$, which is to be filled out with a type $\forall \vec{\alpha}.\, E \Rightarrow (-)$, together with its constructors $\forall_? \cdot S$.
- For each value and stack constructor k, a parameterized version $k_?$ whose parameters are replaced by placeholders to be filled in during type inference and constraint solving.

As to encode costs and potential, the translations involve macros specified below, which are inlined into the transformed program before type-checking with parameters. Those macros take arguments, understood to be terms x, variables st for the token, continuation variables a, and parameters τ. We shall need four such macros:

- *cost*(c, τ): a command that, given a cost τ and a command c, accrues τ and moves on to c;
- *relax*(st, a), a command that, given a token st and a continuation a, returns a *relaxed* token to a, in which an amount of resource R have been allowed to leak from free to used, for some fresh variable R.
- *load*(x, a, st), a command that given a value $x :!A$, a continuation a, and a token st, returns a pair (y, st'), where $y : \mathrm{Pot}_?(A)$ is a version of x charged with some indeterminate potential, and st' is a token in which that potential as been allocated;
- *unload*(x, a, st), dual to *load* that, given a value with potential $x : \mathrm{Pot}_?(A)$, a continuation a, and a token st, returns a pair (y, st'), where $y :!A$ is a version of x without potential, and st' is a token in which that potential as been freed.

7.2 Explicit resource manipulations in programs

Translating types The type-level translation of types merely extends the closures and thunks with quantifiers and tokens. The exponential and fixpoint types will be treated separately, and are therefore omitted in the definition given below:

$$
\begin{array}{rcl}
[\![\mathbb{1}]\!] & = & \mathbb{1} \\
[\![\mathbb{0}]\!] & = & \mathbb{0} \\
[\![A^+ \otimes B^+]\!] & = & [\![A^+]\!] \otimes [\![B^+]\!] \\
[\![A^+ \oplus B^+]\!] & = & \exists_?([\![A^+]\!]) \oplus \exists_?([\![B^+]\!]) \\
[\![\Downarrow A^-]\!] & = & \Downarrow \forall_?(S_? \multimap [\![A^-]\!])
\end{array}
\qquad
\begin{array}{rcl}
[\![\top]\!] & = & \top \\
[\![A^- \,\&\, B^-]\!] & = & \forall_?([\![A^-]\!]) \,\&\, \forall_?([\![B^-]\!]) \\
[\![A^+ \multimap B^-]\!] & = & [\![A^+]\!] \multimap [\![B^-]\!] \\
[\![\Uparrow A^+]\!] & = & \Uparrow \exists_?(S_?. \otimes [\![A^+]\!])
\end{array}
$$

Contexts Γ are translated point-wise. When a variable in Γ is positive, it is translated as-is. If it is negative (i.e. has a computation type), its type is modified as to accounts for an extra token argument of type $S_?$. Indeed, when a negative variable is used, it will be eventually substituted with a computation type which will take control, and must therefore receive a token with it.

$$
\begin{array}{rcl}
[\![(x^+ : A^+), \Gamma]\!] & = & (x^+ : [\![A^+]\!]), [\![\Gamma]\!] \\
[\![(x^- : A^-), \Gamma]\!] & = & (x^- : S_? \multimap [\![A^-]\!]), [\![\Gamma]\!] \\
[\![\emptyset]\!] & = & \emptyset
\end{array}
$$

Continuation contexts Δ which hold a positive continuation $a : A^+$ are upgraded to $a : S_? \otimes A^+$. Indeed, any stack of command will receive a token, and therefore must return it. If the continuation is negative, this means the stack/command does not return the token immediately (as it does not yield control), but waits for a thunk to return it.

$$\begin{array}{rcl} [\![a^+ : A^+]\!] & = & a^+ : S_? \otimes [\![A^+]\!] \\ [\![a^- : A^-]\!] & = & a^- : [\![A^-]\!] \end{array}$$

All programs that perform a reduction or take control are typed with a resource token in scope. This means commands, terms, and environments all have some $st : S_?$ in scope. The case of values and stacks is more subtle, but tractable thanks to polarity. In a positive command $\langle V^+ \parallel S^+ \rangle$, the value wields control to the stack. This means the token should be on the left side, and be substituted into the right side together with the value when the command reduces, as part of a pair. The converse happens in a negative command $\langle V^- \parallel S^- \rangle$: the left side takes control, and the token, from the left side. This is done with an extra argument to the computation. The prototypes for the transforms are therefore as follows:

$$[\![c : (\Gamma \vdash \Delta)]\!] = [\![c]\!] : ([\![\Gamma]\!], st : S_? \vdash [\![\Delta]\!])$$

$$\begin{array}{rcl} [\![\Gamma \vdash t^+ : A^+]\!] & = & [\![\Gamma]\!], st : S_? \vdash [\![t^+]\!] : S_? \otimes [\![A^+]\!] \\ [\![\Gamma \mid S^+ : A^+ \vdash \Delta]\!] & = & [\![\Gamma]\!] \mid [\![S^+]\!] : S_? \otimes [\![A^+]\!] \vdash [\![\Delta]\!] \end{array}$$

$$\begin{array}{rcl} [\![\Gamma \vdash V^- : A^-]\!] & = & [\![\Gamma]\!] \vdash [\![V^-]\!] : S_? \multimap [\![A^+]\!] \\ [\![\Gamma \mid e^- : A^- \vdash \Delta]\!] & = & [\![\Gamma]\!], st : S_? \mid [\![e^-]\!] : S_? \multimap [\![A^-]\!] \vdash [\![\Delta]\!] \end{array}$$

Costs A program specifies its costs via the use of a *cost metric*. Recall that a cost metric μ assigns to each head-reduction $c \triangleright c'$ a cost $\mu(c, c') \in \mathbb{Z}$. In our case, $\triangleright$ is a deterministic reduction, and we write $\mu(c) = \mu(c, c')$ for the only possible reduction from c (or 0 if no such reduction exists). Furthermore, we assume costs can be determined at compile-time. That is, cost is well-defined for commands with free variables as opposed to closed commands. Furthermore, *tick* commands may have been used to incur costs manually. The first step in

the translation of commands is to accrue its cost before moving on:

$$[\![c]\!] = \begin{cases} \mathit{cost}([\![c]\!]_c, \mu(c)) & \text{if } c \text{ is not a tick and } \mu(c) \neq 0 \\ \mathit{cost}([\![c']\!]_c, k) & \text{if } c = \langle \$k \,;\, c' \rangle \\ [\![c]\!]_c & \text{otherwise} \end{cases}$$

Commands, terms, environments The rest of the term-level translation for commands, terms, and environments is merely the simplest one that validates the given prototypes. This specifies the transforms for all commands c, terms t, and environments e. The positive ones are on the left, and negative on the right.

$$\begin{array}{rcl} [\![\langle t^+ \parallel S^+ \rangle]\!]_c & = & \langle [\![t^+]\!]_t \parallel [\![S^+]\!]_e \rangle \\ [\![\mu a^+.c]\!]_t & = & \mu a^+.[\![c]\!] \\ [\![V^+]\!]_t & = & st \otimes [\![V^+]\!] \\ [\![S^+]\!]_e & = & \mu(st \otimes x).\langle x \parallel [\![S^+]\!] \rangle \end{array} \qquad \begin{array}{rcl} [\![\langle V^- \parallel e^- \rangle]\!]_c & = & \langle [\![V^-]\!]_t \parallel [\![e^-]\!]_e \rangle \\ [\![\mu x^-.c]\!]_e & = & \mu x^-.[\![c]\!] \\ [\![S^-]\!]_e & = & st \cdot [\![S^-]\!] \\ [\![V^-]\!]_t & = & \mu(st \cdot a).\langle [\![V^-]\!] \parallel a \rangle \end{array}$$

Variables As to match the new types of positive stacks and negative values, negative variables and positive continuation variables hold the token for the other side.

$$\begin{array}{rcl} [\![x^+]\!] & = & x^+ \\ [\![a^+]\!] & = & \mu x^+.\langle st \otimes x^+ \parallel a \rangle \end{array} \qquad \begin{array}{rcl} [\![a^-]\!] & = & a^- \\ [\![x^-]\!] & = & \mu a^-.\langle x^- \parallel st \cdot a^- \rangle \end{array}$$

Data structures The transformation of constructors for values and stacks is mostly transparent, but care as to be taken around branching. First, parameters describing shapes may take incompatible values between branches. Second, the different clauses involved in pattern-matching need not have the same cost associated to their bodies, but they all receive the same token, which only has one type. The first problem is solved by placing term with sum types under their own placeholders to collect the specific values parameters takes separately in each case (or more precisely, to collect their partial specifications).

Secondly, in order for all those branches to accept the same resources to pay for differing costs, the token is *relaxed* separately in each branch before their bodies take control[2]. This relaxation increases the footprint of each branche by some flexible amount, and is implemented by the *relax* primitive. We shall give more detail about the process of relaxation in the next section, but we can give its use immediately:

[2]Relaxation is not a relaxation of the resource model to a larger, better behaved one, but a relaxation from exact constraints such as (R = R') to inequalities $R \leq R'$.

$$\begin{array}{rcl}
\left[\!\left[k(\overrightarrow{V_i^+})\right]\!\right] & = & \exists_?(k_?(\overrightarrow{[\![V_i^+]\!]})) \\
\left[\!\left[\overrightarrow{\mu(k^i(\vec{x_i}).c_i)}\right]\!\right] & = & \overrightarrow{\mu(\exists_?(k^i_?(\vec{V_i})).\langle \mu a.relax(st,a) \parallel \mu st.[\![c_i]\!]\rangle)} \\
\\
\left[\!\left[k(\overrightarrow{V_i^+}) \cdot S^-\right]\!\right] & = & \forall_? \cdot k_?(\overrightarrow{[\![V_i^+]\!]}) \cdot [\![S^-]\!] \\
\left[\!\left[\overrightarrow{\mu(k^i(\vec{x_i};a_i).c_i)}\right]\!\right] & = & \overrightarrow{\mu(\forall_?(k^i_?(\vec{x_i};a_i)).\langle \mu a.relax(st,a) \parallel \mu st.[\![c_i]\!]\rangle)}
\end{array}$$

When a data or stack constructor is used, it is wrapped in a placeholder which catches the free parameters and equations that occur when it is typed. The placeholder is $\exists_?$ for positive data types, and $\forall_?$ for negative computation types. Note that since the $\forall_?$ placeholder is used in stacks (on the right side), it binds the parameters and equations existentially. On the other side, when a constructor is matched, the placeholder is also matched on, which binds those parameters and equations universally. Lastly, since matching on constructors may induce branching (i.e. case analysis), it is necessary to call *relax* to equate the costs at each branch.

Closures Closures involve a more sophisticated transformation, as they escape their scope of definition. This means the parameters and constraints used to type them at their point of definition may no longer be available at their point of use. Therefore, the parameter-level data needed to infer their footprint must be transported to their point of use. The same goes for the token: there is a token in scope at their point of definition, which pays for the cost of creating them, but they must be used with another token, the one at their point of use.

Closures are positive values $\Downarrow V^- : \Downarrow A^-$, and are thereby transformed without a token in scope when defined, but their inner value $V^- : A^-$ is negative, transformed with one in scope. It suffices to turn A^- into a function $S_? \to A^-$ to get this token.

Likewise, the constraint generated by V^- need not be satisfied as its point of definition, but at their point of use: the inner V^- specifies how it is used, allowing *all* matching environments to interact with it *once the closure is opened*. To encode this behavior, a binder for the placeholder $\forall_?$ and the token is added around the inner value, to store this specification and transport it to the closure's point of use. There, the environment instantiates those parameters and constraint using a transformed stack $\forall_? \cdot S$. This instantiation is done as late as possible, only when the variable bound to the inner value of the closure is used. This specifies the following transformation:

$$\begin{array}{rcl} [\![\Downarrow A^-]\!] & = & \Downarrow \forall_? (S_? \multimap [\![A^-]\!]) \\ [\![\Downarrow V^-]\!] & = & \Downarrow \mu(\forall_? \cdot st \cdot a).\langle [\![V^-]\!] \parallel a \rangle \\ [\![\mu \Downarrow x.c]\!] & = & \mu \Downarrow y.[\![c]\!][\mu a.\langle y \parallel \forall_? \cdot a \rangle / x] \end{array}$$

When the closure is opened ($\mu\Downarrow x.c$), the translated computation must instantiate those parameters and get control over the resource token. This is done by replacing x with a computation that instantiate the parameters using the $\forall_?$ constructor, while the token is already passed to the resulting computation by the general translation of stacks. This guarantees resource manipulations in the closed computation are done only at its call site, but that the information it depends on is transferred from the definition site.

Thunks Thunks are dual to closures, and are treated accordingly. A closure value quantifies its parameters universally and takes a token, and a closure stacks instantiates those parameters and provides a token. Dually, a transformed thunk value instantiates parameters and returns a token when it finishes evaluating, while the stack quantifies universally and captures the token. This translates the fact that a thunk returns *some* value and token, whose type and parameters were determined at its point of definition, but only accessible at its point of use. Once again, the universal quantification occurs as early as possible, and the instantiation as late as possible.

$$\begin{array}{rcl} [\![\Uparrow A^+]\!] & = & \Uparrow \exists_? (S_? \otimes [\![A^+]\!]) \\ [\![\mu \Uparrow a.c]\!] & = & \mu \Uparrow b.[\![c]\!][\mu x.\langle \exists_? (x) \parallel b \rangle / a] \\ [\![\Uparrow \cdot S^+]\!] & = & \Uparrow \cdot \mu \exists_? (st \otimes x).\langle x \parallel [\![S^+]\!] \rangle \end{array}$$

7.3 The case of exponentials, sharing, and fixpoints

Programs involve linear and exponential values, only the latter being shareable. This has fortunate consequences on potential: linear values need not have potential. Indeed, in AARA, potential isn't held by values *per se*, but by variables in scope. In our analysis framework, the token is also always a variable in scope, right next to those variables. We can then meld the potential held by those values and the actual resources in the token. On the other hand, exponential values may close over other exponential values, or more precisely over *some copies* of those values. This means some potential must flow between copies and dependents of exponentials. This sharing of potential does require an explicit handling. This is handled with

the $\mathrm{Pot}_?(A)$ type of exponential values with (yet undetermined) potential, and the *load* and *unload* commands. At type-level, the type $!A$ is transformed to $\mathrm{Pot}_?(A)$, and the term-level transform is the topic of the remainder of this subsection.

$$[\![!A]\!] = \mathrm{Pot}_?(A)$$

Sharing The potential stored inside shared values must be *split* when they are shared, *recovered* when they are deleted, and released when they are demoted to linear values. Recall that, in the System-**L**, sharing is achieved via the command $\langle \sigma; c \rangle$, where σ is a substitution of the shared variables of c to those of available in its surrounding scope. This command is transformed to implement splitting and recovery. Namely, when a variable is omitted, the token first recovers all potential from that variable, and when a variable is shared, the token redistributes potential to each copy. We present here the case of a single shared variable x being shared as n-times as $y_1, \ldots, y_n$ (and omitted if $n = 0$), the general case being handled by iteration of this simpler construct. Precisely, the structural command $\langle [x/y_1, \ldots, x/y_n]; c \rangle$ is transformed as:

$$\begin{aligned}[\![\langle [x/y_1, \ldots, x/y_n]; c \rangle]\!] = &\langle \mu a.\mathit{unload}(x, a, st) \parallel \mu(x, st).\langle [x/y_1, \ldots, x/y_n]; \\ &\langle \mu b_1.\mathit{load}(y_1, b_1, st,) \parallel \mu(y_1, st). \\ &\langle \ldots \\ &\langle \mu b_n.\mathit{load}(y_n, b_n, st) \parallel \mu(y_n, st).c \rangle \ldots \rangle \rangle\end{aligned}$$

In the transformed command, x has type $\mathrm{Pot}_?(A)$. First, x is unloaded, which releases its resources to the token, and provides a raw exponential value. This exponential can then be shared normally. The n fresh copies $y_1, \ldots, y_n$ of x are then charged individually charged with potential. Note that the potential held by x and the one held amongst the n copies may differ: resources may be allocated or freed during the sharing. This is in line with the paradigm of amortized complexity: the difference in potential before and after sharing may be used to derive a lower bound for the footprint. It is only during type checking and constraint solving that the potentials are instantiated and amortized cost of the sharing is determined.

Exponentials Exponential values are built from other exponential values (and only exponential values), and those values may carry potential, which may be carried from the older

values to the new one. But this means some extra work must be done to combine exponentials values and potential. Formally, we need to translate values $V : !A$ such that:

$$\left[\!\left[\overrightarrow{x_i : !B_i} \vdash V : !A\right]\!\right]_t = \overrightarrow{x_i : \mathrm{Pot}_?(B_i)}, st : S_? \vdash V' : \mathrm{Pot}_?(A) \otimes S_?.$$

The type system guarantees that this original $V : !A$ must be a $\mu!a.c$ which closes over the (necessarily exponential) free variables in c. To translate this value, it suffices to unload all those variables before building the exponential, and to reload the potential once the exponential is built. This transformation in shown below, and is implemented by re-using the *load*/*unload* macros to move potential:

$$\begin{aligned}[\![\mu!a.c]\!] = \mu b_0.\langle \mu b_1.\mathit{unload}(x_1, b_1, st) \parallel \mu(x_1, st).\\ \langle \ldots\\ \langle \mu b_n.\mathit{unload}(x_n, b_n, st) \parallel \mu(x_n, st).\\ \langle \mu!a.[\![c]\!] \parallel \mu(st \otimes y).\\ \mathit{load}(y, b, st)\rangle\rangle \ldots\rangle\rangle\end{aligned}$$

On the stack side, $S : !A$ is necessarily some $! \cdot S'$, and must be transformed into some $S : \mathrm{Pot}_?(!A) \otimes S_?$ as to match the value-side. It then suffices to bind the token and exponential, and then to unload and demote the exponential. Namely, this translates as below, where the token is already in scope, as usual for negative stacks:

$$[\![! \cdot S]\!] = \mu x.\langle \mu a.\mathit{unload}(x, a, st) \parallel \mu(y, st).\langle y \otimes st \parallel [\![S]\!]\rangle\rangle$$

This transformation of exponentials and sharing also applies to fixpoints, which always have type $!\mathbf{fix}\, A$ when built. But when the fixpoints are unfolded (layer-by-layer), potential should be involved as to allow for potential to progressively pay for each round of computation.

Fixpoints The body of a fixpoint is copied when it is unrolled, and fixpoints therefore need to be loaded with potential at each layer of the unrolling. To do so, we treat the fixpoint like a closure over itself: as a positive type that holds a negative type (this negative value being the unrolled fixpoint). Following this pattern, a type $\mathbf{fix}\, A$ is transformed into some $\mathbf{fix}\, \forall_?(S_? \multimap A')$. This also has the benefit of enabling calling the fixpoints many times with different parameters, which is essential to encode, for example, structural recursion.

Given the type-level translation, the unfolding stack $\mathbf{fix} \cdot S$ is translated as $\mathbf{fix} \cdot \forall_? \cdot st \cdot [\![S]\!]$, which merely unrolls, instantiates parameters, and passes the token to the freshly unrolled body of the fixpoint.

The translation of on the value side is a more involved: to remain compatible with the original language, self-references must still be exponentials, but the transformed fixpoint has to be hold potential, be polymorphic, and take a token. The translation achieving this is given below.

$$\begin{array}{rcl} [\![\mathbf{fix}\ A^-]\!] & = & \mathbf{fix}\ \forall_?(S_? \multimap [\![A^-]\!]) \\ [\![\mu\mathbf{fix}(a).\langle \mathbf{self} \parallel S^+\rangle]\!] & = & \mu\mathbf{fix}(b).\langle \mathbf{self} \parallel \mu x.\langle \mu(\forall_? \cdot st \cdot a).\langle \mu d.load(x, st, d) \parallel [\![S^+]\!]\rangle \parallel b\rangle\rangle \\ [\![\mathbf{fix} \cdot S^-]\!] & = & \mu(st \otimes x).\langle x \parallel \mathbf{fix} \cdot \forall_? \cdot st \cdot [\![S^-]\!]\rangle \end{array}$$

Note here the sleight of hand: the value $\mu\mathbf{fix}(a).\langle \mathbf{self} \parallel S\rangle$, where a is bound in S, is transformed into some $\mu\mathbf{fix}(b).\langle \mathbf{self} \parallel S'\rangle$, in which the new stack S' returns to a new continuation. This "trick" is essential: the inner stack S (which returns to a) is wrapped with code that handles the parameters and token. The transformed body S is only invoked (with continuation a as required) after the surrounding code has done its work. With this, the transformation scheme is complete. It now remains to see how the primitives we used are implemented in terms of the core parameterized System-**L**.

7.4 Implementing the primitives

The resource token is the core of our implementation of ressource analysis. With this token, we implement cost centers, potentials, and relaxations, which form the high-level specific interface of the analysis. The token, and thereby the whole interface, are already definable in the parameterized type system, as we shall now see.

Definition and primitives We reproduce the definition of the resource token, a parameterized datatype S_U^F (it is a normal datatype to which we give a shorter syntax). It has to **nat** parameters representing a quantities of resources: U denotes an amount of ressources currently in use, and F denotes an amount of currently free resources. During analysis, we use a placeholder $S_?$ for the token type, which gets filled out with specific F and U during type inference.

```
data S^F_U =
  | alloc of S^{F'}_{U'} with K:nat where U' = U + K ∧ F = F' + K
  | free of S^{F'}_{U'} with k:nat where U = U' + K ∧ F' = F + K
end
```

This token supports *allocation* and *release* of resources, written alloc and free respectively. Note that closed value of type S_U^F cannot be built, which ensures that no extra resources can be created by programs, they have to get a token from the environment, as specified by the transformation.

With this type, we can represent allocation and freeing of a variables amount of resources $\varphi(\vec{\alpha})$ using type annotations. Those primitives operations are implemented as commands with a free variable st for the token, a for the continuation, and possibly a parameter argument. They have type:

$$\begin{aligned} \mathit{alloc}(st, a, \tau) &= \langle \mathsf{alloc}_\tau(st) \parallel a \rangle : (\Theta \models \top) \triangleright (st : \mathrm{S}_U^{F+\tau} \vdash a : \mathrm{S}_{U+\tau}^F) \\ \mathit{free}(st, a, \tau) &= \langle \mathsf{free}_\tau(st) \parallel a \rangle : (\Theta \models \top) \triangleright (st : \mathrm{S}_{U+\tau}^F \vdash a : \mathrm{S}_U^{F+\tau}) \end{aligned}$$

Those two commands allow us to define all the primitives in the analysis. Let us begin with the implementation of costs.

Cost centers Cost centers are implemented using the relevant constructor on st, either *alloc* or *free*. The computed value for the amount of resources moved l is stored in the constructors, providing a history of the resource usage as they are progressively stacked. The *cost* primitive is implemented as transformed as so:

$$\mathit{cost}(c, k) = \begin{cases} \langle \mu a.\mathit{alloc}(st, a, k) \parallel \mu st.c \rangle & \text{if } k > 0 \\ \langle \mu a.\mathit{free}(st, a, -k) \parallel \mu st.c \rangle & \text{if } k < 0 \\ c & \text{if } k = 0 \end{cases}$$

Using type annotations and parameterized types, it is also possible to encode non-constant costs. For example, consider natural integers are defined by a Church encoding as:

```
data Int(α:nat) =
  | z with α = 0
  | s of Int(α') with α':nat with α = α' + 1
end
```

Consider, also an abstract type $\mathsf{Array}(A, \alpha)$ of arrays of elements of type A with size $\alpha : \mathbf{nat}$. A function that allocates an array of size α and initializes it with copies of a shared value of

type $!A$ can be typed as:

$$\mathit{array\_init} : \mathsf{Int}(\alpha) \multimap !A \multimap \Uparrow\mathsf{Array}(!A, \alpha)$$

which takes an integer n and a value $V : !A$ and returns an array of length n containing V in all fields. This function can be wrapped with a cost $\varphi(\alpha)$, where φ is a variable of sort **nat** $\to$ **nat**, which can be user-defined or merely declared. This gives cost-aware function *array_init'*, defined below using the high-level *alloc* primitive and the original *array_init*:

$$\begin{aligned}
&\mathit{array_init'} : \forall\alpha.\, \mathrm{S}_U^{F\,+\,p(\alpha)} \multimap \mathsf{Int}(\alpha) \multimap !A \multimap \Uparrow(\mathrm{S}^F_{U\,+\,p(\alpha)} \otimes \mathsf{Array}(!A, \alpha)) \\
&= \mu(\mathsf{spec}_\alpha \cdot st \cdot (n : \mathsf{Int}(\alpha)) \cdot x \cdot \Uparrow \cdot a). \\
&\quad \langle \mu b.\mathit{alloc}(st, b, p(\alpha)) \parallel \mu st. \\
&\quad\ \langle \mathit{array_init} \parallel n \cdot x \cdot \Uparrow \cdot \mu y. \\
&\quad\quad \langle (st, y) \parallel a \rangle \rangle \rangle
\end{aligned}$$

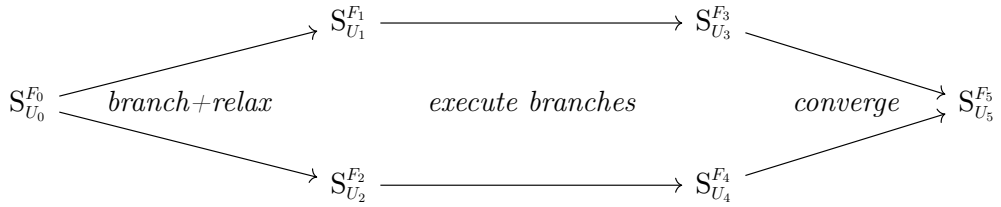
Costs center can therefore also be implemented in the case where costs vary depending on input values, which makes our analysis more general: variable costs for user-defined primitives can be specified, and inlined during constraint solving.

Relaxation Relaxation produces *approximations* of resource usage, which are required when control flow is not deterministic. For example, a program with a $\mathsf{if} \ldots \mathsf{then} \ldots \mathsf{else} \ldots$ expression has a different cost depending on the branch taken in the conditional. It is, general, not possible to separately bound the footprints all possible code paths within a program. It may even not be possible to known in advance which of them would incur a larger cost, as one branch's cost may overtake the other depending on the values involved. This makes the token's parameters hard or impossible to instantiates when the two paths rejoin. In general, any use of pattern-matching with many clauses will cause those problems.

Relaxation allows the resource footprints of different code paths to be rejoined under the same type. This is done by increasing the footprint of each path, as a first step after they branch. The many flexible increment can then be chosen to have the many footprints line up. The relevant primitive is obtained from the *alloc* command using a free parameter variable R : **res**. The parameter R being free allows the amount of relaxation to be flexible, being determined later on under a global optimization. Its implementation is:

$$\mathit{relax}(st, a) = \mathit{alloc}(st, a, R) : (\Theta, R \models \top) \triangleright (st : \mathrm{S}_U^{F\,+\,R} \vdash a : \mathrm{S}^F_{U\,+\,R})$$

As an example, of the inner workings of relaxation, consider a program with two code paths, and a resource token being passed through it. When both paths begin to diverge, they both have control over same token $\mathrm{S}_{U_0}^{F_0}$. But when they end, they return some possibly distinct tokens, $\mathrm{S}_{U_3}^{F_3}$ and $\mathrm{S}_{U_4}^{F_4}$ respectively. But suppose they are each relaxed beforehand to give tokens $\mathrm{S}_{U_1}^{F_1}$ and $\mathrm{S}_{U_2}^{F_2}$ respectively, and then unified into some token $\mathrm{S}_{U_5}^{F_5}$. This is illustrated below.



Those tokens are constrained by the type system to respect the following equations, where R_i is the amount of the resources relaxed when going from $\mathrm{S}_{F_0}^{U_0}$ to $\mathrm{S}_{U_i}^{F_i}$:

$$U_i + F_i = U_j + F_j \quad (i, j \in [0, 5])$$

$$\begin{aligned} U_1 \geq U_0 \qquad & F_1 \leq F_0 \\ U_2 \geq U_0 \qquad & F_2 \leq F_0 \\ U_3 = U_4 = U_5 \quad & F_3 = F_4 = F_5 \\ R_1 = U_1 - U_0 \quad & R_2 = U_2 - U_0 \end{aligned}$$

In short, the right half of the diagram induces exact correspondences between states, and the left one partially specifies the initial state required to cover for either evaluation path. Solving this system for U_0, F_0 and the two relaxed quantities R_1 and R_2, in terms of the other quantities, we obtain:

$$\begin{aligned} U_0 \leq \min(U_1, U_2) \qquad & F_5 \geq \max(F_1, F_2) \\ R_1 \geq \max(0, U_1 - U_2) \quad & R_2 \geq \max(0, U_2 - U_1) \end{aligned}$$

We see that the relaxed approximations of used resources U_0 at the beginning of the program is lower than the used resources at beginning of both branches U_1 and U_2, and likewise the approximated amount of free resources F_0 is larger than its two real counterparts F_1 and F_2. That is, branching allocates enough resources to cover both footprints. The original tokenis therefore sufficient to cover either branch, and $U_0 + F_0$ is a sound footprint.

The amount of resources wasted by performing this approximation is the value R' defined below. It is null when the resources relaxed in the low-cost branch are exactly enough to cover the high-cost one. In this case, the relaxation still provides exact results.

$$R' = \max(R_1, R_2) - \max(U_1 - U_2, U_2 - U_1).$$

The relaxed token $\mathrm{S}_{U_0}^{F_0}$ can therefore be explicitly written as $\mathrm{S}_{\min(U_1, U_2) + R'}^{\max(F_1, F_2) - R'}$. The implicit encoding of those expressions using constraints does not require the min/max operators or the use of the non-total subtraction on natural integers.

If R' is zero (which is desirable), the relaxation merely picks the higher footprint of the two, and does so without needing to choose one of the path. This is essential, as which of the two has the highest cost may change depending on inputs, may not be statically determinable, or simply too hard to compute in practice. Furthermore, adding the waste terms gives a safe approximation, even in the case where maximum/minimum of the symbolic expressions involved cannot be statically computed. This waste term enables solvers to replace the complex min/max term by a simpler but less precise term if exact bounds cannot be determined.

Relaxation thereby allows bounds for the resource footprints of programs to be computed when control flow diverges between branches, and to recover a sound approximation of resource footprints when two branches converge back.

Potential Potentials allow resources to be allocated in the approximated footprint before they are actually used by the program, and associated to values whose access unlocks said resources. This allows resource footprints to be smoothed: in a queue, for example, potential can be associated to elements on their insertion, and used to pay for their processing as the queue is consumed. As we saw in section 2.1, this allows proving that the cost of the queue over its lifetime depends only on the number of insertions, as opposed to depending on a specific sequence of insertions and accesses.

In many resource analyses, resources are associated to variables in scope, which allows for a natural implementation of potential, but requires a guarantee that resources are soundly manipulated when those variables are accessed. In our setup, all resources are held in the resource token, and their total quantity are preserved by definition. This directly guarantees soundness, but implies that values cannot hold potential directly. Instead, we represent a

value holding a potential P by a value that, when given control of the token, releases P resources from its *used* section to its *free* section. With this setup, having potential is not having resources, but having the ability to *release* resources. As such, resources can remain within the token.

More precisely, consider a value $V : !A(\overrightarrow{\tau})$ parameterized by $\overrightarrow{\tau}$, that we want to bundle with a polynomial potential $P = \varphi(\overrightarrow{\tau})$. To do so, we define the type $\mathrm{Pot}(A, \varphi, \overrightarrow{\tau})$ as a function that frees $\varphi(\overrightarrow{\tau})$ resources when called and returns a shared value without potential. This gives the following type definition, which replaces the $\mathrm{Pot}_?(A)$ placeholder during type inference:

$$\mathrm{Pot}(A, \varphi, \overrightarrow{\tau}) = \Downarrow \forall F, U.\ \mathrm{S}^F_{U\,+\,\varphi(\tau)} \multimap \Uparrow \left(\mathrm{S}^{F\,+\,\varphi(\tau)}_U \otimes\, !A(\overrightarrow{\tau})\right)$$

This type can be introduced by passing a value $A(\overrightarrow{\tau})$ without potential and a token with at least $\varphi(\overrightarrow{\tau})$ resources, and returns a $\mathrm{Pot}(A, p, \overrightarrow{\tau})$ and a token where those resources are used. It then can then be eliminated by a calling the function underlying the $\mathrm{Pot}(A, p, \overrightarrow{\tau})$, which recovers the shared value and the $\varphi(\overrightarrow{\tau})$ resources, now ready to be used. The amount of potential $\varphi(\overrightarrow{\tau})$ is not specified during translation. Instead, a fresh function variable is introduced for φ, to be instantiated and optimized later on during constraint solving.

We can now implement the two last primitives. The first one, $\mathit{load}(x, a, st)$ takes a value x and a token st, and returns a potential holding-value bearing potential $\varphi(\overrightarrow{\tau})$, where φ is fresh, and a token where those $\varphi(\overrightarrow{\tau})$ resources are now allocated. On the other side, $\mathit{unload}(x, a, st)$ releases all the potential held in x to the token, and returns both to a:

$$\mathit{load}(x, a, st) : (\Theta \models \top) \triangleright \left(x : A(\overrightarrow{\tau}), st : \mathrm{S}^{F\,+\,p(\overrightarrow{\tau})}_U \vdash a : \mathrm{Pot}(A, p, \overrightarrow{\tau}) \otimes \mathrm{S}^F_{U\,+\,p(\overrightarrow{\tau})}\right)$$
$$\mathit{unload}(x, a, st) : (\Theta \models \top) \triangleright \left(x : \mathrm{Pot}(A, p, \overrightarrow{\tau}), st : \mathrm{S}^F_{U\,+\,p(\overrightarrow{\tau})} \vdash a : A(\overrightarrow{\tau}) \otimes \mathrm{S}^{F\,+\,p(\overrightarrow{\tau})}_U\right)$$

This concludes our presentation of the resource manipulation constructs in programs. Remains to show how data structures and computations are extended with parameters to create shape-aware types, and how those newly-introduced parameters are handled.

7.5 Soundness

Consider a closed program $c : (\vdash \star : \Uparrow \mathbb{1})$. After token explicitation, it becomes

$$c' : (\Theta \models C) \triangleright (st : \mathrm{S}^F_U \vdash \Uparrow(\mathrm{S}^{F'}_{U'} \otimes \mathbb{1})).$$

If the head-reduction of c has a cost profile with a prefix $(k_i)_{i \leq N}$, then this $(k_i)_{i \leq N}$ is also a prefix of the token trace of c' under reduction-under-context. Therefore, if $(\Theta \vdash C)$ is satisfiable, then, in any model $\Theta \models C$, the sum $U + F$ is larger than $\Sigma_{i \leq N} k_i$. As such, $F + U$ is larger than the maximum prefix of the trace held in the token, and finally larger than the token footprint of the original command c.

This can be proved by induction on the size of the prefix, and using the soundness theorem of the previous chapter. During the induction step, a case analysis on the reduction rule applied last allows us to reason case-wise. In each case, this one reduction step before translation corresponds to one-or-many steps in the translated command, which, by construction of our translation, pushes the right cost onto the token, plus some eventual slack. Therefore, by definition of the token trace, and thanks to the soundness theorem, the compile-time constraint C_i at step i provides, if it has any model at all, a sound upper bounds for the cost at step i. And since we have $C_i \Rightarrow C_{i+1}$ for all i, if the the original constraint $C_0 = C$ has any model, it provides a sound upper bound for the costs at each step.

7.6 Generating analyzable datatypes

In this section, we use the resource manipulation setup we developed to implement AARA for simply-typed algebraic data-types. We do so by giving a relevant parameterization to each ADT which recovers the enumerative combinatorics that AARA used its indices for. Namely, given an ADT definition as the first one below, we transform it into the second one below. Note that while data types may be recursive, the only kind of recursion supported by AARA is the one presented below. For example, the data type $T(A) = k$ **of** List$(T(A))$ is not recursive as far as AARA is concerned, as $T(A)$ only appears under other type constructors.

The new parameters $\vec{\alpha}$ count the number of some predefined patterns in values of that type, and those counts are kept accurate as the ADT is inductively built through the use of algebraic relations.

data $K(\vec{A}:\vec{\mathbf{m}}) =$
  | k_1 **of** B_1 $*$ $K(\vec{A})$ $*$ $\cdots$ $*$ $K(A)$ *(* n_1 terms *)*
  ...
  | k_m **of** B_m $*$ $K(\vec{A})$ $*$ $\cdots$ $*$ $K(A)$ *(* n_m terms *)*
end

```
data K(A⃗:m⃗)( α⃗:nat⃗ ) =
  | k_1 of B_1 * K(A⃗) (β_{1,1}⃗) * ··· * K(A) (β_{1,n_1}⃗)  where β_{1,1}⃗, ..., β_{1,n_1}⃗ with E_1
  ...
  | k_m of B_m * K(A⃗) (β_{m,1}⃗) * ··· * K(A) (β_{m,n_m}⃗)  where β_{m,1}⃗, ..., β_{m,n_m}⃗ with E_m
end
```

Linear AARA In linear AARA, indices count the number of each constructor $k_1, \ldots, k_m$ in a value V of type $K(\vec{A})$. For a given value $V = k_i(W, \vec{V'})$, this is the sum of the number of constructor in each V', plus one for the constructor k_i to account for the top-level constructor. Note that it excludes the constructors held "indirectly" through the value W of type B_i.

Formally, the vector α is composed m natural integers, and the value of the i^{th} component α_i denotes the number of constructors k_i in V, excluding those within the non-recursive arguments of type B_i.

For a given constructor k_j, the equation E_j enforces that the sum of the numbers of constructors k_i in the recursive arguments is equal to the total number of k_i constructors in V (possibly decreased by one if $i = j$). That is, with δ_j the vector whose i^{th} is one if $i = j$ and zero otherwise,

$$E_j = \left(\vec{\alpha} = \sum_{i=i}^{n_j} \overrightarrow{\beta_{j,i}} + \vec{\delta_j} \right).$$

For a typical example, binary trees would end up bearing to parameters, one counting the number of leaves (λ), and one counting the number of inner nodes (ν). This would give the overall parameterized type declaration:

```
data BT(A,B:+) (λ,ν:nat) =
  | Leaf of A where (λ = 1) ∧ (ν = 0)
  | Node of B * BT(A,B)(λ_l, ν_l) * BT(A,B)(λ_r, ν_r)
    where λ_l,λ_r,ν_l,ν_r:nat
    with (λ = λ_l + λ_r) ∧ (ν = ν_l + ν_r + 1)
end
```

Multivariate AARA A finer parameterization of binary trees consists in counting not only the number of nodes, but also the number of chains of nodes up to a certain length. For

example, the number of (leaf, ancestor) pairs that the tree holds, or the number of triplets of inner nodes (x_1, x_2, x_3) in which x_i is an ancestor of x_{i+1}.

For example, when counting chains of two nodes, there are four possible patterns to look for: inner/inner, inner/leaf, leaf/leaf, and leaf/inner. Those last two can obviously never occur, so let us focus on the inner/inner and inner/leaf pairs. To count them, one must also count the number of leafs and inner nodes as previously done.

This gives four parameters in total, λ and ν again, and two new parameters α^{NN} and α^{NL} for inner/inner and inner/leaf pairs respectively. When a tree consists of a single leaf, all the parameters are zero expect λ, and when two trees are packed in an inner node, each leaf of those two trees creates an inner/leaf pair, and each inner node an inner/inner pair. This leads to the following parameterization:

data BT(A,B:+)(λ,ν, α^{NN},α^{NL} : **nat**) =
| Leaf **of** A **where** $(\lambda = 1) \wedge (\nu = 0)$ $\wedge(\alpha^{NN} = 0) \wedge (\alpha^{NL} = 0)$
| Node **of** B $*$ BT(A,B)(λ_l,ν_l, α_l^{NN},α_l^{NL}) $*$ BT(A,B)(λ_r,ν_r, α_r^{NN},α_r^{NL})
where λ_l,λ_r,ν_l,ν_r, α_l^{NN},α_r^{NN},α_l^{NL},α_r^{NL} : **nat**
with $(\lambda = \lambda_l + \lambda_r) \wedge (\nu = \nu_l + \nu_r + 1)$
$\wedge(\alpha^{NN} = \alpha_l^{NN} + \alpha_r^{NN} + \nu_l + \nu_r)$
$\wedge(\alpha^{NL} = \alpha_l^{NL} + \alpha_r^{NL} + \lambda_l + \lambda_r)$
end

In the general case, the parameterization induced by multivariate AARA is based on *strings of constructors*. For a given ADT T defined as having constructors $k_1, \ldots, k_n$, such a string denotes a pattern in values of type T made of a chain of constructors. Fix a maximum length L for those strings, and write I the set of all (possibly empty) strings of length at most L with symbols $k_1, \ldots, k_n n$. The parameters of T are then a family $(\alpha_l)_{l \in L}$. Each constructor definition is parameterized as follows:

data $K(\vec{A})$($(\alpha_l)_{l \in L}$) =
...
| k **of** B $*$ $K(\vec{A})$ $((\beta_{1,l})_{l \in L})$ $*$ $\cdots$ $*$ $K(\vec{A})$ $((\beta_{n,l})_{l \in L})$ **where** $(\beta_{j,l})_{j \leq n_i, l \in L}$ **with** E
...
end

The parameter α_l counts of the number of string of constructors l in a value, and $\beta_{i,l'}$ counts the number of strings of constructors l' present in the i^{th} recursive argument. Finally, the

equation E relates the two. Formally, E is the conjunction, for each string $l \in L$, of:

$$\alpha_l = \begin{cases} 1 & \text{if } |l| = 0 \\ \sum_{j \leq n} \beta_{j,l} + \beta_{j,l'} & \text{if } 0 < |l| < N \text{ and } l = kl' \\ \sum_{j \leq n} \beta_{j,l} & \text{if } |l| = N \end{cases}$$

That is, the number of strings l in the value is the sum of the number of strings l in each argument, but also, if l begins with the current constructor k (that is, $l = kl'$), of the number of strings l' in each argument. That is, are counted not only the strings in each argument, but also the strings *completed* by the current constructor.

In all encoding of multivariate AARA as above, the equations added during parameterization are all *substitutions* (that is, of the form $\alpha = \tau$ for some variable α and term τ). Furthermore, they are together *acyclic* (parameters counting strings of length n are substituted for terms with parameters for strings of length $n - 1$). This will be important during implementation, as we'll see in section 8.7.

Potential Now that we can endow ADTs with parameters according to the AARA methodology, we can define AARA potential as a linear combination of the indices. Consider a ADT $K(\vec{A})(\vec{\alpha})$ with AARA parameters as above, and its potential-bearing version $\text{Pot}(K(\vec{A}), \varphi, \vec{\alpha})$. The potential function φ for AARA is a linear combination of the $\vec{\alpha}$, with coefficients $\vec{\beta}$. Those $\vec{\beta}$ are declared at top-level, and are subject to a linear optimization problem providing their values. Namely, the full AARA encoding $K_{AARA}(\vec{A})$ of $K(A)$ is:

$$K_{AARA}(\vec{A}) = \exists \vec{\alpha}.\text{Pot}(K(\vec{A}), \varphi, \vec{\alpha}), \text{ where } \varphi(\vec{\alpha}) = \sum_i \beta_i \alpha_i$$

This encoding is applied recursively to all non-recursive arguments of each type to endow each value with potential. Note that the location of potential and resource information differs between standard AARA implementation and ours. In the former, potential and resource info is held in the local scope, within each variables. In our system, resources are held in the resource token, which is unique, and resource information and potential is held within values, possibly deep in their type.

Iterators Finally, our potential formalism can be used to create iterators that imitate RAML's support for structural recursion. While RAML synthesizes those structural recursions

from `let rec` defined function, we only present here the structural iterators themselves, leaving the compilation of the former to the latter for future work.

Recall that, given an ADT definition as seen below, an isomorphism between the ADT and its unfolded form as:

$$K(\vec{A}) \simeq \bigoplus_{i=1}^{n} \left(B_i \otimes K(\vec{A})^{m_i} \right).$$

Using the properties of parameterized negation in intuitionistic logic and linear logic, a function of type $K(\vec{A}) \multimap C$ can be equivalently represented as a computation of type

$$\neg K(\vec{A}) := !\left(\mathop{\&}_{i=1}^{n} (B_i \multimap (\Downarrow C)^{m_i} \multimap C) \right)$$

For example, for linked list, such an universal destructor will take the form of the well-know *fold* combinator. Namely, a fold for a list of type $!\mathrm{List}(A)$ computing a C is defined by an initial computation of type $!\Downarrow C$ and an iteration step $!(A \multimap (\Downarrow C) \multimap C)$. This matches precisely our definition:

$$\begin{aligned} \neg\mathrm{List}(A) &\simeq !\left((\mathbb{1} \multimap C) \,\&\, (A \multimap \Downarrow C \multimap C)\right) \\ &\simeq\ (!C) \otimes !(A \multimap \Downarrow C \multimap C) \end{aligned}$$

Performing the same operation on the parameter aware encoding of an ADT $!K(A)(\vec{\alpha})$ provides the AARA destructor on the nose. An iteration takes as arguments an data structure $\exists\vec{\beta}.!K(A)(\beta)$ and a fold $\neg\vec{\exists}\alpha.!K(A)(\beta)) \simeq \forall\vec{\alpha}.\neg K(A)(\vec{\alpha})$ as arguments, and therefore has type

$$!\left(\mathop{\&}_{i=1}^{n} \left(\forall\vec{\alpha}, \overrightarrow{\beta_{i,j}}.\ E_i \Rightarrow B_i \multimap \Downarrow C(\overrightarrow{\beta_{i,1}}) \multimap \ldots \multimap \Downarrow C(\overrightarrow{\beta_{i,m_i}}) \multimap C(\vec{\alpha}) \right) \right) \multimap \forall\vec{\alpha}.\ K(\vec{A})(\vec{\alpha}) \multimap C(\vec{\alpha})$$

Making this iteration resource-aware requires only making the computation $C(\alpha)$ pass resources along. This means setting, for some *linear* functions p, q and return type $D(\alpha)$:

$$C(\vec{\alpha}) = \forall F, U.\ \mathrm{S}^{F+p(\vec{\alpha})}_{U+q(\vec{\alpha})} \multimap \Uparrow D(\alpha) \otimes \mathrm{S}^{F+q(\vec{\alpha})}_{U+p(\vec{\alpha})}$$

This choice of computation with cost allows us to recover the waterline and final resource usage from the before and after tokens. To do so, set $U = F = 0$. In this case, the before

token has type $\mathrm{S}^{p(\vec{\alpha})}_{q(\vec{\alpha})}$ and the after token has type $\mathrm{S}^{q(\vec{\alpha})}_{p(\vec{\alpha})}$. This shows that the total amount of resources in the system is $(p+q)(\vec{\alpha})$ and that the change in used resources is $(p-q)(\vec{\alpha})$.

For brevity, we do not spell out the full general type, but give an example for lists. We take for our example a degree two AARA analysis. This means the type $L(A)$ of lists of type A is parameterized with three integers $\alpha^{\varepsilon}, \alpha^{C}, \alpha^{CC}$ denoting, respectively, the number of empty lists of constructors (always one), the number of *nil* constructor N, the number of *cons* constructor C, and the number of ordered pairs of two lists elements (strings CC). Note that we have $\alpha^{CC} = \frac{1}{2}\alpha^{C}(\alpha^{C}-1)$ and $\alpha^{\varepsilon} = 1$

With this parameterization of lists, the cost of a computation is determined by two tuples $\vec{p}$ and $\vec{q}$, which determine, together a with a parameterization $\vec{\alpha}$, a resource amount given by the dot product $\vec{p}\cdot\vec{\alpha} = p^{\varepsilon}\alpha^{\varepsilon} + p^{C}\alpha^{C} + p^{CC}\alpha^{CC}$. Each step of the list fold then takes a resource token having $\mathrm{S}^{F+\vec{p}\cdot\vec{\alpha}}_{U+\vec{q}\cdot\vec{\alpha}}$ for some $\vec{p}, \vec{q}, \vec{\alpha}$ and returns a token having $\mathrm{S}^{F+\vec{q}\cdot\vec{\alpha}}_{U+\vec{p}\cdot\vec{\alpha}}$ resources.

$$
\begin{array}{ll}
!\Downarrow\forall\vec{p},\vec{q},\vec{\alpha}. & \textit{For all costs } \vec{p},\ \vec{q} \textit{ and params. } \vec{\alpha}, \\
\left(\forall F,U.\ \mathrm{S}^{F+p^{\varepsilon}}_{U+q^{\varepsilon}} \multimap \Uparrow\left(D(0)\otimes \mathrm{S}^{F+q^{\varepsilon}}_{U+p^{\varepsilon}}\right)\right) & \textit{Turn an initial value,} \\
\multimap\ \Big(!\Downarrow\forall\vec{\beta},\vec{\gamma}. & \textit{and iterator turning...} \\
\quad (\beta^{\varepsilon}=\gamma^{\varepsilon}\wedge\beta^{C}+1=\gamma^{C}\wedge\beta^{CC}+\beta^{C}=\gamma^{CC})\Rightarrow & \\
\quad A & \textit{an element...} \\
\quad \multimap\Downarrow\left(\forall F,U.\ \mathrm{S}^{F+\vec{p}\cdot\vec{\beta}}_{U+\vec{q}\cdot\vec{\beta}} \multimap \Uparrow\left(D(\vec{\beta})\otimes \mathrm{S}^{F+\vec{q}\cdot\vec{\beta}}_{U+\vec{p}\cdot\vec{\beta}}\right)\right) & \textit{and the previous value...} \\
\quad \multimap\left(\forall F,U.\ \mathrm{S}^{F+\vec{p}\cdot\vec{\gamma}}_{U+\vec{q}\cdot\vec{\gamma}} \multimap \Uparrow\left(D(\vec{\gamma})\otimes \mathrm{S}^{F+\vec{q}\cdot\vec{\gamma}}_{U+\vec{p}\cdot\vec{\gamma}}\right)\right)\Big) & \textit{into the net one,} \\
\multimap L(A)(\vec{\alpha}) & \textit{take a list of lentgh } \vec{\alpha}, \\
\multimap \forall F,U.\ \mathrm{S}^{F+\vec{p}\cdot\vec{\alpha}}_{U+\vec{q}\cdot\vec{\alpha}} \multimap \Uparrow\left(D(\vec{\alpha})\otimes \mathrm{S}^{F+\vec{q}\cdot\vec{\alpha}}_{U+\vec{p}\cdot\vec{\alpha}}\right) & \textit{into the final value.}
\end{array}
$$

With this type, the initial value provided by argument sets the value of p^{ε} and q^{ε}, and the iterator provides the rest of the coefficients of $\vec{p}$ and $\vec{q}$. Finally, the values of α^{ε}, α^{C} and α^{CC} are related by the equations $\alpha^{\varepsilon} = 1$ and $\alpha^{CC} = \frac{1}{2}\alpha^{C}(\alpha^{C}-1)$.

Putting all of this together, the waterline cost $C(n)$ of the processing of a list is of length n is the value of the polynomial $p+q$ evaluated at the values of α for a list of length n is a

polynomial $C(n) = c_2n^2 + c_1n + c_0$. Indeed,

$$\begin{aligned}
C(n) &= (\vec{p} + \vec{q}) \cdot \vec{\alpha} \\
&= (p^{CC} + q^{CC})\alpha^{CC} + (p^C + q^C)\alpha^C + (p^\varepsilon + q^\varepsilon)\alpha^\varepsilon \\
&= \frac{1}{2}(p^{CC} + q^{CC})n(n-1) + (p^C + q^C)n + (p^\varepsilon + q^\varepsilon) \\
&= \frac{1}{2}(p^{CC} + q^{CC})n^2 + (p^C + q^C - \frac{1}{2}p^{CC} - \frac{1}{2}q^{CC})n + (p^\varepsilon + q^\varepsilon) \\
&= c_2n^2 + c_1n + c_0
\end{aligned}$$

We can compare this result with the similar one obtained in paragraph 2.6 to check the validity of the approach. Indeed, using the AARA formalism, the complexity of the `insert` sort function defined as a fold is estimated as:

$$\frac{1}{2}q_2n^2 + (q_1 - \frac{3}{2}q_2)n + q_0$$

We can then match the coefficients as:

$$\begin{aligned}
q_2 &= p^{CC} + q^{CC} = c_2 \\
q_1 &= p^C + q^C + p^{CC} + q^{CC} = c_1 + c_2 \\
q_0 &= p^\varepsilon + q^\varepsilon = c_0
\end{aligned}$$

We can see that the parameters (c_2, c_1, c_0) we assigned to the fold with our paramterization, and the parameters (q_2, q_1, q_0) derived for that same fold by the original AARA method, are merely a linear transformation away from each other. We therefore implemented AARA using another base for the linear space of potential function, which achieves the same results.

7.7 Closing words

Using the parameter system we added to **L**, we were able to develop, in this section, an automated resource analysis. Using an automatic rewrite phase to-and-from the machine, we can automatically enrich CBPV programs with resource-passing. This requires no extension to the general-purpose parameter system. Resource-passing is implemented as a call-by-push-value effect, in which focusing was essential to respect the ordering of resource manipulation. The case of non-linear constructs required special care as to recover the potential formalism

of amortized analysis. Finally, we showed how it was possible to link resource usage to the structure of algebraic datatypes *à la* multivariate AARA.

This ends the second part of this thesis focused on the machine, its enrichment with parameters and the implementation of resource analysis with those parameters. Thanks to its operational precision and flexibility provided by CBPV, it can be used as a target for resource analysis for a variety of functional languages, and cover features such as monadic effect systems in source languages. The rest of this thesis is dedicated to presenting our implementation of **L**, and comparing our method with other formalisms in the state of the art.

Chapter 8

Implementation

System-**L**, its abstract machine and parameterized type systems are not merely theoretical objects, but are implementable and can interface with solvers and proof assistants. This is what we show with our implementation, *AutoBill*. In this section, we first present the AutoBill compilation and analysis pipeline in section 8.1, and give a first example of input in machine language and output in section 8.2. Then, we describe its type inference procedure in section 8.3, and its parameter and constraint inference procedure in section 8.4. This will be the occasion to see some first results from solvers. Section 8.5 is dedicated to a discussion about the theoretical complexity of parameter inference. Finally, we discuss how AutoBill transforms constraints into optimization problems in section 8.6, and provide an example of those optimization capabilities in section 8.7.

8.1 Presentation

AutoBill is an open-source compiler and type-checker for parameterized System-**L**, written in OCaml and licensed under the *GNU Public License version 3*. It provides a command-line interface and a web interface, with both a server-and-client architecture and a fully client-side architecture that includes an embedded solver. Its public repository can be found at the following link[1].

[1] https://gitlab.lip6.fr/suzanneh/autobill has the freshest code, but may expire in 2026, https://github.com/Polybulle/Autobill is currently a mirror and will host long-term development.

In its standalone command-line executable form, it can compile and typecheck the three languages present in the frontend: an ML-like language with monadic effects, a CBPV-ML λ-calculus, and a continuation-passing term language mapping directly to System-**L**. From nay of those inputs, a type-elaborated and optimized System-**L** program can be obtained. Furthermore, a first-order parameter constraint can be obtained (under several formats), whose satisfiability suffices to guarantee the program's well-typedness in the presence of parameters. Finally, in the case of polynomial indices, a linear integer programming problem can be generated, from which a closed form can be derived for an arbitrary "objective" polynomial. We strove to not make AutoBill rely on large software dependency: compilation only requires the *Dune* build system and the *Merlin* parser generator, relying otherwise on the OCaml standard library.

The web interface wraps around this command-line tool to provide easy iterative development. It has been built by Brahima Dibassi, Zeid Fazazi, and Yukai Luo as part of a project course during their first year of Master's at *Sorbonne Université*. It is built with *React.js*, with AutoBill and solvers running on a distant server, or with AutoBill and the *MiniZinc* solver and optimizer running directly client-side.

Pipeline The compilation pipeline of AutoBill is presented in figure 8.1. This figure contains the main data structures, simplified compilation steps, and annotations denoting input, output, and tracing capabilities.

The AutoBill frontend can input and output untyped code written in either the effectful ML-style language, the pure CBPV-ML language, and System-**L** code. The compilation of imperative blocks into monadic primitive was implemented by Noé Weeks as part of a summer internship. Hard-coded implementations of those primitives in CBPV-ML are embedded in the frontend. The rest of the ML language is translated according the CBV embedding into CBPV. The CBPV-ML language is then compiled to the linear, continuation-passing System-**L**.

Once this frontend pipeline is completed and the untyped **L** code has been generated, the analysis proper can begin. First, the code is internalized, i.e. each variable is checked for scope and given a globally unique name. Then, sort-checking occurs. Prelude definitions and declarations are checked for validity, and in the program itself, the polarity of variables

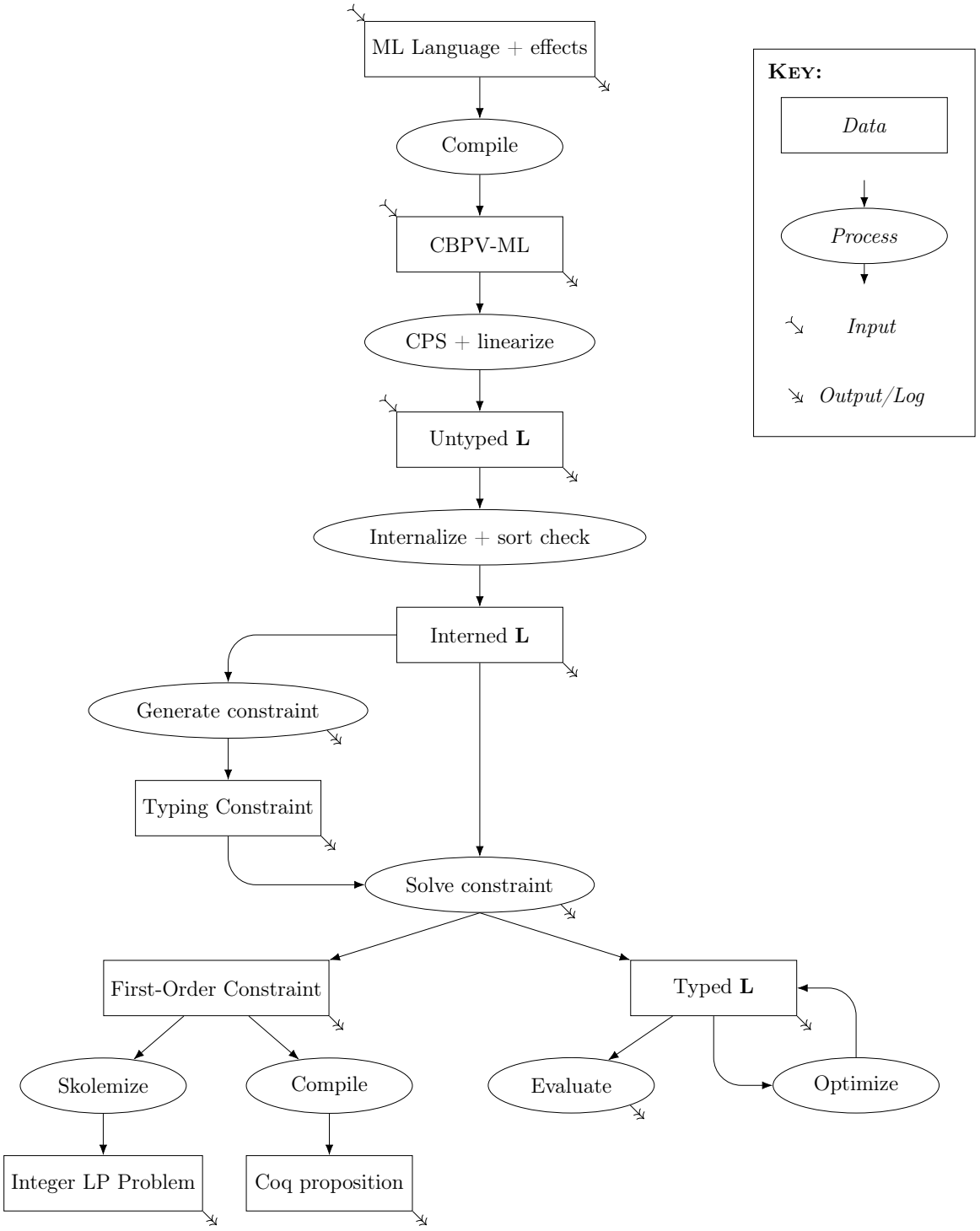


Figure 8.1: Compilation pipeline of *AutoBill*

and sort of type annotations is checked. This produces an interned **L** program and a cached prelude for the next steps.

AutoBill the proceeds with type and parameter inference. This will produce a type-checked program with elaborated type information on each expression, and produces a first-order constraint on the parameter involved in the program. To do this, a typing constraint is generated from the program, whose satisfiability implies the well-typedness of the program, and whose models provide elaborated parameters. We will look more closely at this process in the next section. The typing constraint is not *fully* solved by this step, as it contains parameters of arbitrary first-order sorts, whose corresponding theory is not known by AutoBill. As such, the typing constraint is solved *up to* a first order parameter constraint, and the parameter variables used in this constraint also occur in the elaborated program's types. As such, once the constraint is generated, it is solved externally and independently of the program to assess its satisfiability and eventually produce models.

Once the typed program and parameter constraint are generated, they can be process separately. The program itself can be evaluated, which reduces it to weak-head normal form if possible (the program's evaluation may not terminate due to the presence of fixpoints). It can also be optimized through normalization-by-evaluation: some of the reduction rules under contexts are used to safely create a normal form without risk of divergence or code size increase. To do this, all rules are iterated, except the fixpoint we only apply once per fixpoint in the original program, and the structural rule, which is triggered only when it would not cause copying.

Separately, the first-order parameter constraint generated from the typing constraint is processed. First, it is compacted using rules such as $\exists x, y.\exists z.C \simeq \exists x, y, z.P$ and $C \wedge \top \simeq C$. It can then be exported to either a custom S-expression encoding, or as a standalone Coq proposition, or, if it only contains parameters of **nat** sort, simplified to a conjunction of polynomial (in)equations, and turned into a linear programming problem over the rationals. The processing of the first-order constraint will also be explored further in the remainder of the chapter.

The nature of AutoBill as an workbench to validate and experiment on methods for resource analysis means it deviates from and sometimes lags behind the theory we developed. We wish to highlight the following differences, which we find significant enough to be worthy of mention. This also serves as a short "future work" section about the implementation:

- First is the effect system for call-by-value ML and its interaction the linearization procedure. As it stands, it is only possible to translate ML programs into the machine by ignoring the linearity conditions.

- Second, the inference of general monotypes of sorts $\vec{\mathbf{p}} \to \pm$ is not yet supported. Of course, parameterized type constructors, are fully supported and can take parameter arguments. But inference of arbitrary monotypes is more subtle. Indeed, given two types of sort $\vec{\mathbf{p}} \to \pm$, determining whether they are unifiable, and under which conditions, requires at least a form of higher-order unification, and possibly knowledge of the theories the sorts $\vec{\mathbf{p}}$ used to interpret them. This limits the expressiveness of the automated parameter inference.

- For similar reasons, the placeholders $\forall_?$ and $\exists_?$ is currently not sufficiently expressive as implemented. This is due to the difficulty of extracting which parameter information held under the placeholder can be lifted above it. This difficulty is of the same nature are those met in the implementations of generalized algebraic datatypes. See [66] and [68] for more details on how Haskell and OCaml respectively handle the same problem.

- We also have gone further than the core language presented in this manuscript. For example, the languages implemented by *AutoBill* contain many quality-of-life improvement and syntactic sugar that simplify writing programs. This includes many ways to write commands that resemble *let*-bindings for values and continuation, and the ability to write expressions of the form $k(\vec{t})$ as opposed to merely values $k(\vec{V})$.

- Furthermore, the parameter constraint used by Autobill also support higher-order logic, that is, parameters of non-base sorts. This includes multivariate polynomials, which are parameters of sort $(\mathbf{nat}, \ldots, \mathbf{nat}) \to \mathbf{nat}$.

8.2 A first example

To introduce the System-**L**syntax used for in Autobill, let us present the code of a simple *swap* function that take two arguments x, y and returns the tuple (y, x). The listing of the source program, the typechecker output, and the evaluated program, are presented in figure 8.2.

```
1  decl type A : +
2  decl type B : +
3  decl val x0 : A
4  decl val y0 : B
5  cmd ret a =
6  val swap =
7    match this.call(x,y).ret(b) ->
8      cmd
9      val = match this.thunk().ret(c) -> tuple(y,x).ret(c)
10     stk = this.ret(b)
11 in
12   swap.call(x0,y0).thunk().ret(a)
```

(a) Input program (`swap.bill`)

```
13 decl type A_21 : +
14 decl type B_22 : +
15 decl val+ x0_37 : A_21
16 decl val+ y0_39 : B_22
17 cmd+ anon_43 ret a_42 : (B_22 * A_21) =
18   cmd- : (Fun A_21 B_22 -> (Thunk (B_22 * A_21))) val =
19     match this.call(x_46 : A_21, y_48 : B_22).ret(b_50 : (Thunk (B_22 * A_21))) ->
20     cmd- : (Thunk (B_22 * A_21)) val =
21       match this.thunk().ret(c_57 : (B_22 * A_21)) ->
22       cmd+ : (B_22 * A_21) val =
23         tuple(y_48, x_46)
24       stk =
25         this.ret(c_57)
26     stk =
27       this.ret(b_50)
28   stk =
29     this.bind- swap_76 : (Fun A_21 B_22 -> (Thunk (B_22 * A_21))) ->
30     cmd- : (Fun A_21 B_22 -> (Thunk (B_22 * A_21))) val =
31       swap_76
32     stk =
33       this.call(x0_37, y0_39).thunk().ret(a_42)
```

(b) After typechecking (`autobill -L swap.bill -t`)

```
34 decl type A_21 : +
35 decl type B_22 : +
36 decl val+ x0_37 : A_21
37 decl val+ y0_39 : B_22
38 cmd+ anon_43 ret a_42 : (B_22 * A_21) =
39   tuple(y0_39, x0_37).ret(a_42)
```

(c) After evaluation (`autobill -L swap.bill -r`)

Figure 8.2: The swap function in *Autobill*

Lines 1-4 in the input program declare two fresh types and variables of those types. In general, new sorts, parameters, types, and values can be declared. The rest of the program, from line 5 onward, is made of a command returning its result to a free continuation variable `a`. This command has syntax `val` x `=` V `in` c, which is a convenient shorthand for $\langle V \parallel \mu x.c \rangle$. The value defined here is the swap function, which is defined by case analysis on the stack on line 7. All stack pattern start with the `this` keyword, followed by a stack constructor (here, `call`), and terminated by a continuation variable (here, `a` in `ret(a)`). The body of the function is a command spanning lines 7-10. It is a command written in long form (the syntax is `cmd val =` t `stk =` e, standing for $\langle t \parallel e \rangle$). The term used in this command is the thunk that, when forced with a continuation `c`, returns the tuple to c (syntax `tuple(y,x).ret(c)`). The continuation stack in the commands body immediately returns the thunk when the function is called. Finally, on line 12, the function is called with the declared variables as arguments. This is once again a command, this time with syntax $V \,.\, S$, using the `swap` function as value, and the sequence of constructors `call` and `thunk` as stack, terminated by the free continuation variable `a` to return the result as the end value of the program.

The second listing is the output of type inference on this file. We note the following changes. First, each identifier is suffixed with a globally unique, auto-incremented counter added during internalization to prevent scoping issues when rewriting syntax trees during processing. For example, the declared variable `x0` of type `A` in line 3 has become, in line 15, the variable `x0_37` of type `A_21`. Secondly, some keywords, such as `cmd` and `bind`, have been annotated with a sign symbol giving their polarity. Also, all commands have been converted to long form. Finally, the type of all commands, variables, and stack variables have been annotated. It is now possible to see that the inferred type of the swap function is, as expected, `(Fun A_21 B_22 -> (Thunk (B_22 * A_21)))`. The last listing is the same file again, this time after evaluation. All declarations are preserved, and the local function `swap` is now absent, having been substituted. The end result is a command that, as expected, returns the tuple with the swapped variables.

After this first presentation of Autobill, we can focus our attention to the inference algorithms that underlie its features. Let us start now with type inference.

8.3 Type inference

The type inference algorithm we implemented is an extension of Pottier & Remy's constraint-based implementation of Hindley-Milner type inference (HM) described in their monograph published in [67]. We deviate from this specification by adding type variables interpreted in arbitrary first-order theories, as opposed to having only variables interpreted in the theory of "uninterpreted terms" as is traditional HM type inference. Which theory is used is fixed by the sort of the variables. Types with positive and negative polarities are uninterpreted terms, and parameters have some unknown theory (unknown at type inference time, that is).

This addition of unspecified theories for some sorts impedes type inference in the following three ways: first, without a specified theory, unification of two parameter expressions is not possible. Second is *determination*: given a unification problem $u = e[\overrightarrow{v}]$ between a variable u and a type-level term e involving other variables $\overrightarrow{v}$, is there a subset $\overrightarrow{w}$ of $\overrightarrow{v}$ that, for any assignment $\overrightarrow{w_0}$, there is a single assignment of $\overrightarrow{v} \setminus \overrightarrow{w}$? solution ? In that case, $\overrightarrow{w}$ *determines* $\overrightarrow{v}$ and u. With uninterpreted terms, the determination problem is readily solvable. But it needs not be solvable for a given first-order theory, and is absolutely undoable in without knowing the theory in which the variables are interpreted. Third is the *lifting* problem. Namely, given a constraint

$$\exists \overrightarrow{v_0}.\forall \overrightarrow{v_1}.\exists w.\overrightarrow{e = e'} \Rightarrow C,$$

is it sound to lift w over the universal quantifier into the $\overrightarrow{v_0}$? In other words, does a fixed assignment of the existential variables $\overrightarrow{v_0}$ determines a single assignment of w ? In that case, we say w *is dominated* by the variables $\overrightarrow{v_0}$. Once again, algorithms exist to solve the domination problem in constant amortized times for uninterpreted terms, but no such solution is available in general.

Those problems are solved by delegating the work of treating the parameters to a second constraint, the *parameter constraint*. With this setup, solving the typing constraint doesn't completely guarantee correctness, but only does so *relatively to the parameter constraint*. It is obtained using standard type inference algorithms, but stopping when unifying, establishing determination, and lifting variable with parameter sorts. This eliminates the type variables with base sort as would a normal HM type inference algorithm, but leaves a remainder in the form of the parameter constraint. In that sense, ours is a partial type inference procedure. Thankfully, this remaining parameter constraint is expressible in first-order logic as opposed to

the full typing constraint. A possible avenue for improvement would be to run an SMT solver alongside the type inference procedure, as to provide inferred terms for the parameters at the same time as the type-level terms of base sorts. This appears to be doable by instrumenting the Z3 solver.

Furthermore, this partial type inference procedure does allow for the elaboration of types in the original program, once again up to parameters. This means the types of all expressions and variables can be computed up to the values of parameters. The parameters appearing in those types are replaced with placeholder variables that also appear in the parameter constraint, and sound assignment of parameters in the constraint gives sound assignments in the type annotations. Therefore, while full type inference is not possible in the presence of arbitrary parameters, it does lead to a partially elaborated program and corresponding parameter constraint, which we argue is an optimal situation given the presence of arbitrary first-order theories in the parameter language. Nevertheless, this issue does not occur when the constraints involved are generated by AARA-parameterized programs, which admit solving procedures.

8.4 Resources and bounding

Before moving on to the handling of parameters within Autobill, it is worth spending some time with a minimal example in which resources are modeled and bounded. Figure 8.3 shows a program defining a resource token, the output constraint Autobill produces on such a file, and the output of an external solver on this constraint producing a bound on the footprint described

The first listing shows the content of the input Autobill file. It begins with the definition of the token parameterized datatype, `Token F U`. The parameters F and U represent the integral amount of free and used resources respectively. This type possesses a single constructor `cost` encoding a manipulation of resources. It takes as argument another token of type `Token F2 U2`, moves A resources from free to used, and B resources from used to free. Note the syntax `cost<...>` to specify parameters introduced by the constructors, while the `with ...` used in the formal calculus is used to specify equations using a comma-separated list.

```
data Token (F:nat) (U:nat) =
  | cost<A:nat, B:nat, F2:nat, U2:nat>(Token F2 U2)
    with (Add F A) = (Add F2 B), (Add U B) = (Add U2 A)

decl type N : nat

goal N degree 0

decl val token : Token N Z

cmd ret a = cost<One,Z>(token).ret(a)
```

(a) Input file (`token.bill`)

```
(declare-const T_286 Int) (assert (<= 0 T_286))
(declare-const T_285 Int) (assert (<= 0 T_285))
(declare-const T_284 Int) (assert (<= 0 T_284))

(assert (= 0 (+ (+ T_285 1) (* -1 T_284))))
(assert (= 0 (+ T_286 -1)))

(assert (<= 0 T_284)) (minimize T_284)

(check-sat)
(echo "Goal is N() = T_284")
(get-objectives)
```

(b) Optimization constraint (`autobill -L token.bill -o token.smt2 -lpsmt`)

```
sat
Goal is N() = T_284
(objectives
 (T_284 1)
)
```

(c) Result of the optimization (`z3 -smt2 token.smt2`)

Figure 8.3: An initial example of resource bounding

Once this token type is defined, an objective amount of resource to minimize is defined. It is declared as an integer parameter `N`, and set as the objective using the `goal` directive. This directive can occur at most once per program, and has syntax `goal ...degree ...`. It takes as argument the name of a type constructor taking parameters of sort **nat** as argument and returning a `nat`, understood as a polynomial. Here, using `N:nat` as an argument, it is understood as a polynomial with zero argument. The `degree` directive sets the maximal degree this polynomial is assumed to have, and must be a positive-or-null integer.

Finally, with this objective set, an initial token value of type `Token N Z` is declared. It then has N free initial resources and zero initial used ones (`Z` is the Autobill syntax for the type-level integer zero). This token is used by the command terminating the file, which manipulates resources by applying the `cost<One,Z>(...)` constructor to the token. The two parameter constants `One` and `Z` are annotated to specify the amount of resource moved. Namely, one resource is freed and none are allocated. Note that only two parameters are passed to the constructor, but that it is defined to take four parameter arguments. When such partial application of parameter is used, Autobill generates placeholders parameters for the missing arguments. Given that initial token has `N` free resources and zero used, and that one is allocated and none freed, valid assignment for `N` are those with $\texttt{N} > 1$.

The second listing is the constraint generated by Autobill as a file in the *SMT-LIB 2* format. It is expressed in the quantifier-free fragment of first-order logic, with conjunctions, implications, and the integers with signature $(0, 1, +, -, *, =\leq)$. In lines 1-3, the variables of the constraints are defined and asserted to all be non-negative. This manual assertion is required since most solvers implement solving for $\mathbb{Z}$ but not $\mathbb{N}$. Then, in lines 5-6, the constraint itself is encoded as a sequence of two linear expressions which must be null. In line 8, the optimization problem is defined by asserting that the variable `T_284` representing N be minimized. The rest of the file starts the optimization process and sets up its output. The result of the optimization is shown on the last listing of the figure. Its first line, whose content is always `sat` or `unsat`, describes whether an assignment of the program's parameter N exists, i.e. whether the program is well-typed in the parameterized type system. In this case, it is. The second line gives the formula for the goal polynomial N to be found. Here, it is of degree zero, i.e. a constant. Finally, the rest of the output gives the value of the coefficients that provides a minimal valid polynomial that fits the program, here $N = 1$.

With this first example in tow, let us now explore how Autobill handles parameters, how constraints are generated, and how they are solved.

8.5 Parameter inference

The compilation scheme we introduced in chapter 7, which bridges the simply typed machine with a resource metric and the parameterized machine with a token, is implemented on top of the Hindley-Milner type system. Recall that we introduce placeholders $\forall_?(A)$ and $\exists_?(B)$, which need to be filled with some $\forall\vec{\alpha}.(E \Rightarrow A)$ and $\exists\vec{\alpha}.(E \wedge B)$ respectively to make programs parameter-aware.

Making programs parameter-aware can vastly increase its order of polymorphism. Even functions which are fully monomorphic in ML languages often may be parameter-polymorphic. For example, the monomorphic ML function $sum : List(Int) \to Int$ should become a polymorphic System-**L** function of type

$$sum : \forall n.List(Int, n) \to Int$$

Otherwise, it could only be called with lists of a fixed size. Should we want to give ML values parameterized types while preserving their ability to be reused on argument of varying sizes, we must quantify their parameters. This means polymorphism is ubiquitous in type system with parameters.

Placing inference point During the type checking, placeholders $\forall_?$ and $\exists_?$ were inserted to collect those quantified parameters, and the constraints that specify them. The question then becomes, where should those placeholders be inserted ? And, to which placeholder should free parameters variables and equations be assigned ?

The grammar of types with placeholders allows $\forall_?$ to be placed above any negative type, and $\exists_?$ over any positive one. But in the compilation scheme, they were only ever used below shifts $\Downarrow$, $\Uparrow$, **fix**, and under sum types $\otimes$ and &. The question is then, why ?

Focusing provides an answer to this question, by giving a normal form to types in which quantifiers can be given a canonical position. This property can be formalized as such: consider a positive type A^+ or a negative type A^- defined by the following two grammars, where B^+ and B^- are well-sorted applications of parameters to a monotype, and E are well-sorted

simple constraints:

$$A^+, B^+ ::= \Downarrow B^- \mid !B^- \mid K^+(\vec{\alpha}) \mid A^+ \otimes A^+ \mid A^+ \oplus A^+ \mid \mathbb{0} \mid \mathbb{1} \mid E \wedge A^+ \mid \exists \alpha. A^+$$
$$A^-, B^- ::= \Uparrow B^+ \mid K^-(\vec{\alpha}) \mid B^+ \multimap A^- \mid A^- \& A^- \mid \top \mid E \Rightarrow A^- \mid \forall \alpha. A^-$$

Then, there exists types isomorphic to A^+ and A^- respectively, of with the following *prenex form* (this which obtained by inductively lifting quantifiers, then sums). In this decomposition, the basic blocks B^+ are either closures, exponentials, fixpoint, or type constructors, and B^- are thunks or type constructors.

$$A^+ \simeq \exists \vec{\alpha}. E \wedge \bigoplus_{i=1}^{n} \bigotimes_{j=1}^{m_i} B^+_{i,j}$$
$$A^- \simeq \forall \vec{\alpha}. E \Rightarrow \mathop{\&}_{i=1}^{n} \left(\left(\bigotimes_{j=1}^{m_i} B^+_{i,j} \right) \multimap B^-_i \right)$$

Those prenex forms allow quantifiers and logical manipulation of simple constraints to be lifted across types, and accumulate them under $\Downarrow$, $\Uparrow$, ! and **fix**. Those fours types corresponds both to changes in polarity and breaks in normal control flow (closures and thunks are delocalized, exponentials are shared, and fixpoint loops). Finally, the placeholders can be lowered through the additive connectives $\oplus$ and & without loss of generality, to create types matching the parameterized data/computation types we already defined. This means that, overall, any positive type A^+ (resp. negative type A^-) is isomorphic to some normal form:

$$A^+ \simeq \bigoplus_{i=1}^{n} \left(\exists \vec{\alpha}_i. E_i \wedge \bigotimes_{j=1}^{m_i} B^+_{i,j} \right)$$
$$A^- \simeq \mathop{\&}_{i=1}^{n} \left(\forall \vec{\alpha}. E \Rightarrow \left(\bigotimes_{j=1}^{m_i} B^+_{i,j} \right) \multimap B^-_i \right),$$

where the types $B^+_{i,j}$ (resp. B^-_i) are either units, predefined data constructors, a closure, exponential, or fixpoint (resp. a unit, predefined computation type, or a thunk). This normal form enables us to lift quantifier as much as possible without risk of invalidating them through branching where they don't apply, or of lifting them past a shift, i.e. from a call site to a definition site.

This will nevertheless not completely solve the problem of quantification of parameters in resource analysis. Indeed, nested quantification, as required for higher-order programs with parameters, does not allow for all-encompassing solutions.

The problem with parameter polymorphism The need to quantify over parameters causes a problem in the case of types involving both constraint-manipulating types and shifts ($\Downarrow$, $\Uparrow$, or **fix**), such as higher-order functions or locally defined functions. Indeed, the parameters created in those cases could be quantified either on the type of the inner function, or be lifted to be quantified in the type of the outer expression. This will have consequences in terms of typability, as not quantifying parameters that vary at call sites may cause the parameterized program to fail typechecking. Unfortunately, this problem of deciding, in the context of nested quantifiers, where should type variables be bound, is undecidable. This was shown by Wells [80]in the context of type inference of *System F*, a λ-calculus with higher-order polymorphism discovered independently by Girard [28] and Reynolds [70].

The design of statically-typed functional languages must deal with this impossibility in their treatment of polymorphism (universal quantification) and we should mention two approaches as to illuminate our treatment of automated parameterization, its rationale, and limitations. First, in the *Hindley-Milner* type system (HM)[32, 61], values themselves may not have a quantified type, only variables. Only binders such as `let` x `=` e `in` e' may produce quantifiers. Namely, the inferred, quantifier-free type A of e produces the inferred type $\forall\vec{B}.\, A$ for x, where$\vec{B} = \mathbf{fv}(A)$. Any subsequent use of x in e' is then unified with $A[\vec{B'}/\vec{B}]$, for some fresh type variables $\vec{A'}$.

The second approach, taken by *Haskell*, allows *higher-ranked types*, that is to say types with quantifier which cannot be simplified to type schemes, such as functions with polymorphic arguments. This is based on a system of required annotations on bound variables and some sub-expressions, and locally uses type inference to reduce the quantity of annotations required, as explained in detail in [42]. Despite the added type-level expressiveness, the demands made on the user are significant.

Indeed, while we are sympathetic to the trade-off proposed by this method in the case where *types* are annotated, we cannot demand it of users of a resource analyser, on which *parameters* would have to be annotated. Nevertheless, on the practical side, the all-important case where

parameters are polynomials with integer coefficients, those limitations can be partially worked around using quantifier elimination, which we shall do here.

Utility of higher-ranked types Higher-rank types do appear to boost expressiveness in machine code, allowing, for example, iteration primitives to be implemented with an annotated, resource-aware type. For example, a `map` function on lists, whose mapping function takes an element $A(k)$ at index k, returns a new element $B(k)$, while moving $p(k)$ from free to used and $q(k)$ resources from used to free, would have the overall type:

$$\Downarrow\forall n.\left(\forall k.A(k) \multimap \mathrm{S}^{F+p(k)}_{U+q(k)} \multimap \Uparrow\mathrm{S}^{F+q(k)}_{U+p(k)} \otimes B(k)\right) \multimap \mathit{IList}(A,n) \multimap \mathrm{S}^{F+p'(n)}_{U+q'(n)} \multimap \Uparrow\mathrm{S}^{F+p'(n)}_{U+q'(n)} \otimes \mathit{IList}(B,n)$$

In this type, A and B both have sort $\mathbf{nat} \to +$, the (yet impossible) indexed list *IList* has sort $(\mathbf{nat} \to +) \to \mathbf{nat} \to +$, and the polynomial p' (resp. q') is defined as the function $\lambda n. \sum_{k=0}^{n-1} p(k)$ (resp. $q(k)$), using a "sum" type constructor of sort $(\mathbf{nat} \to \mathbf{nat}) \to (\mathbf{nat} \to \mathbf{nat})$. Higher-ranked types are here used to defined the indexed element types A and B, the data type *IList*, and the "sum" constructor, and enable building an arguably natural type to the map function when the cost of mapping each element depends on its position in the list.

Since higher-ranked types do allow for increased expressivity, but require a much more involved treatment of parameter inference, there is an obvious point of tension, which, once again, does not allow for fully-satisfying solution. It is our opinion that, nevertheless, their implementation in AutoBill could power novel and finer-grained approaches to AARA analyses in the future.

8.6 Constraint solving

Once the type inference procedure is complete, an elaborated program and a first-order parameter constraint are generated. The program's well-typedness is relative to the satisfiability of this constraint, and its models provide assignments to the program's parameters. Autobill relies on external solvers to find those models and generate optimal assignments for the parameters, the same way RAML uses a linear programming library to derive its bounds. We now discuss the processing of this constraint and our use of external solvers.

General case The grammar of parameters constraints is given by the following fragment of multi-sorted first-order logic. The actual implementation differs through the use of n-ary conjunctions in constraint C and removing the binary conjunction in E.

$$\langle E \rangle ::= \top \mid t = t \mid R(\overrightarrow{t}) \mid E \wedge E$$

$$\langle C \rangle ::= \top \mid \bot \mid C \wedge C \mid \exists \overrightarrow{\alpha}.\, E \wedge C \mid \forall \overrightarrow{\alpha}. \exists \overrightarrow{\alpha}.\, E \Rightarrow E \wedge C$$

Before doing serious work on the constraint, some prepossessing is done to reduce its size without changing its semantics (satisfiability and models). In short, this is done by lifting existential quantifier until they meet a universal one (which absorbs them, as occurs HM type inference, see [67]), or reach the top-level (at which point they accumulate there). Conjunctions are also compacted using their associative, unit, and absorption law. Note that it is not sound to fuse universal quantifier in this grammar of first-order logic, as it might move some part of the consequence of an implication out of scope from its hypothesis.

At this point, in the presence of user-defined sorts or unknown theories, no more progress can be made. The first-order constraint can nevertheless be exported. We support export to the Coq proof assistant, which can be used to prove satisfiabilty, but not optimize a resource bound. Just as well, the constraint can be exported in the *SMT-LIB 2* format to use any off-the-shelf SMT solver to check satisfiability, and, in the positive case, obtain non-optimized bounds.

Programs with quantified parameters We exemplify this by checking a parameterized-program, in the number of iteration of an iterator is checked for correctness without any annotations in its implementation. This is shown in figure 8.4.

In this program, a parameterized list datatype (`List`) is defined together with its formal dual (`CoList`). Each value of that dual type defines and "step" computation which consumes and element of the list (called `coCons`), and a "finalize" computation that produces a final return value from the iterator's hidden state (called `coNil`). Together with a list, this defines a fold. A recursive function called `fold` combines the list and iterator, the same way the well-known `fold_left` function combines a list, initial value, and accumulation function to produce a final result.

```
data List (A : +) (N : nat) =
  | nil() with N = Z
  | cons<M : nat>(A, List A M) with N = (Add M One)

comput CoList (A : +) (B : -) (N : nat) =
  | this.coNil().ret(B) with N = Z
  | this.coCons<M:nat>(A).ret(Fold A B M) with N = (Add M One)

comput Fold (A:+) (B:-) =
 // Replace this 'N' with a 'Z' to break typecheck ----↓
 this.fold_it<N:nat>(List A N, Closure Lin (CoList A B N)).ret(B)

val fold = match this.fix().ret(a) ->
  self.bind fold ->
    cmd
    stk = this.ret(a)
    val = match this.fold_it(l,f).ret(b) ->
      l.match
      | nil() -> f.unbox(Lin).coNil().ret(b)
      | cons(h,t) ->
        val f = box(Lin) c -> f.unbox(Lin).coCons(h).ret(c) in
        fold.fix().fold_it(t,f).ret(b)
      end
```

Figure 8.4: Safe resource-aware iterators

At the beginning of the program, line 1-7, we declare the parameterized list type and its dual `CoList`. Each method in `CoList` is the dual of a constructor in the original `List` type. The parameterization bears implies that one can only call the `coNil` method on a value of type `CoList A B Z` (once again, Z is zero), and only call `coCons` on those of type `CoList A B N` with $\mathtt{N} \geq 1$. As such, to process a list of type `List A N` with a fold of type `CoList A B N`: not only must the type `A` of the elements match, but their numbers as well. In that sense, is it a minimal example of parameter-awareness and parameter-quantification.

The relation between the length of the list and iteration count is set through a type constructor defined lines 9-11, that acts as parameter-polymorphic type declaration for `fold`. This introduction of a new type constructor is used only to set a point at which to quantify parameters. Indeed, the type defined therein, `Fold`, is merely a computation that, given a list and iterator with the same parameter, returns the same result. The *raison d'être* of this type constructor is the fresh parameter variable N in scope within its constructor, which is the common parameter of the two argument. This introduction is governed by the use of the `fold_it` constructor the `Fold` type defines.

The rest of the file hosts the definition of the polymorphic computation that, given a list and iterator of the same "size", combines them to produce the result, on lines 13-23. From this point onward, no more type and parameter annotations are required. The fold is defined by induction on the list, as one would expect a `fold_left` function to be in a functional language. Lines 13-16 merely set up the recursion using a fixpoint and continuation passing. The "blackboard" syntax $\mu\mathbf{fix}(a).\langle\mathbf{self} \parallel S\rangle$ used to define fixpoint is translated in textual syntax as `this.fix().ret(a) -> self.some_stack`, where `some_stack` is the aforementioned S.

On line 17, the `fold_it` constructor is matched on, which introduces the parameter N in its definition universally, and binds the list l and fold f, both of length N. On line 18, the list is matched on, in which introduces another parameter `N'` universally, under the assumption that $\mathtt{N} = \mathtt{N}'$ due to the type of lists. The two clauses of that match introduce either the hypothesis $\mathtt{N}' = 0$ or $\mathtt{N} = \mathtt{M} + 1$, for another universally quantified M to which the hypothesis apply. The same process continues for the rest of the code, introducing parameters universally (resp. existentially) and equations as hypotheses (resp. proof obligations) when constructors are matched (resp. used to build a value or stack).

```
1  // generated by: autobill -L fold.bill -z
2
3  (assert
4   (forall ((T_75 Int) (T_76 Int) (T_240 Int) (T_241 Int))
5    (exists ((T_283 Int))
6     (=> (and (= T_75 T_241) (= T_76 T_240))
7      (and
8       (and
9        (= T_240 T_283))
10      (and
11       (forall ((T_100 Int) (T_284 Int) (T_555 Int) (T_556 Int))
12        (exists ((T_116 Int) (T_134 Int) (T_135 Int) (T_345 Int) (T_346 Int)
13                 (T_388 Int) (T_389 Int) (T_550 Int) (T_551 Int) (T_552 Int)
14                 (T_553 Int) (T_554 Int))
15         (=> (and (= T_100 T_284) (= T_283 (+ T_284 1)) (= T_555 (+ T_284 1))
16                  (= T_556 1))
17          (and
18           (and
19            (= T_241 T_345)
20            (= T_345 (+ T_346 1))
21
22     (...lines elided here...)
23
24 (check-sat)
25 (get-model)
```

Figure 8.5: Safe resource aware iterator: output

SMT solving *Z3*[22] is an open-source solver developed by *MicroSoft* research. It supports a combination of various theories, notably (for our purposes) first-order linear arithmetic. While it cannot optimize models of formulas in this theory, it still can be used for verification by proving that a parameter constraint is satisfiable or not.

The constraint generated by that program is shown in figure 8.5. The quantifier `(forall ...(T_240) ...)` on line 2 introduces `N`, which is unified with the variable `T_283` representing `N'` on line 7. Then, The next quantifier on line 9 introduces M, under the hypothesis `(= T_283 (+ T_284 1))` specified on line 13. Likewise, all other variables or equations of the original programs are incorporated into the constraint. When this constraint is fed into the *Z3* solver, it validates the program's parameterization with a laconic `sat`. On the other hand,

if the length of the fold is changed to no longer match the length of the list, then the solving will fail. For example, replacing the length with `Z` as suggested by the comment on line 10 of the listing will cause *Z3* to reject the constraint with an equally laconic `unsat`.

As this shows, the parameterized system of the **L**-machine, as implemented in Autobill, can be used to prove the well-typedness of parameter-aware programs through the use of SMT solving. To obtain resource bounds on the other hand, another post-processing is required. Indeed, to our knowledge, finding optimal models of constraints such as the one we above cannot be done by readily available solvers due to the presence of universal quantifiers. As such, in order to implement AARA analyses in Autobill, more processing of constraints is required.

8.7 Optimizing parameters

When implementing AARA, going from a parameter constraint to a closed-form bound can be fully automated. To obtain a constraint amenable to optimization, we shall use some simplifying assumptions and normalization algorithms. First, some logically irrelevant simplifications are used to reduce the size of the constraint, such as removing unused variables and compacting n-ary conjunctions. Those operations are performed as routine during the processing as to obtain human-readable output constraints. First, the constraint is simplified by identifying some equalities to use as substitutions. This will lead either to a constraint which can be made into an equivalent linear programming problem, or one which cannot be processed further.

To determine which equalities are amenable to this transformation, we define the *rank* of a variable to be the number of alternations of $\forall$ and $\exists$ binders from the root of the formula up to and including the quantifier binding it. The rank of term is then the minimum of the ranks of its free variables, or $+\infty$ otherwise. This determines whether an equality can be made into a substitution: $t = t'$ can become a substitution iff t equals some variable x whose rank is smaller or equal to that of t', at which point it generates the substitution $[x \mapsto t']$. If the case of an equality $x = y$ with both variables of the same rank, we choose witch one to substitute arbitrarily.

If an equality in an implication is of the form $t = t'$ for non-variable terms t and t', then we should decide whether to give up on processing it further. The criterion we choose for this is that, to obtain a LP problem, none of the free variables in either t or t' should be bound by an universal quantifier. The presence of universal variables does not mean the constraint overall isn't equivalent to an LP problem, but our choice of test was a compromise between success rate and simplicity of implementation. We shall explain this later once the rest of the processing is introduced.

Skolemisation The next step of processing (see fig. 8.1) is to remove the formula's quantifiers to obtain one in which all variables are parameters are declared globally. The first step is to remove existential quantifiers through *skolemisation*, that is, replacing all existential variables with calls to a fresh function.

Formally, in a top-down fashion, every existential variable α is transformed into a term $f_\alpha(\vec{\beta})$, where f_α is a fresh function symbol, and the $\vec{\beta}$ are all variable in scope at the point of the existential binder $\exists\alpha.$, which, by hypothesis, must all be universal variables. This is done recursively over all existential binders, and the fresh function symbols are introduced to global scope. After skolemisation, the first-order constraint now fits the following grammar, where the only free variables are the new function symbols f_α and the universal variables β.

$$\langle C \rangle ::= \exists \vec{f}.\, \forall \vec{\beta}.\, C'$$

$$\langle C' \rangle ::= E \mid \bot \mid C' \wedge C' \mid E \Rightarrow C'$$

$$\langle E \rangle ::= \top \mid t = t' \mid t \leq t'$$

Instantiating functions Then, we remove the function symbols from the constraint by instantiating them. Doing so is sound but not complete. Indeed, we shall instantiate them with a polynomial term of fixed degree, but function symbols may be modeled by a higher-degree polynomial, or even a non-polynomial function such as a logarithm or exponential. In such cases, our solving procedure may fail to satisfy a constraint, or over-approximate the functions. Nevertheless, it can be pragmatically argued that the costs, sizes, and potentials involved in an algorithm have a tendency not use high-degree terms (for example, it is quite rare to see algorithms with complexities $\Theta(n^{100})$). Let us therefore pick a maximal degree N. Given a

function symbol f_α of arity n, we assign to it the *free polynomial* on n variables and degree N. This polynomial is defined in the standard manner, which we recall here.

Let $\mathbf{X} = (X_1, \ldots, X_n)$ be a family of n variables and a sequence $\mathbf{i} = (i_1, \ldots, i_n)$ a sequence of n natural integers, which we call a multi-index. We define the *degree* $\deg(\mathbf{i})$ of $\mathbf{i}$ is the sum $i_1 + \cdots + i_n$, and the *monomial* $\mathbf{X}^\mathbf{i}$ as the product $X_1^{i_1} \ldots X_n^{i_n}$. Finally, let $(a_\mathbf{i})_{\deg(\mathbf{i}) \leq N}$ be a family of integer variables indexed by all multi-indices of degree at most N. The *free polynomial on variables $\boldsymbol{X}$ of degree n* is the sum $\sum_{\deg(\mathbf{i}) \leq n} a_\mathbf{i} \mathbf{X}^\mathbf{i}$.

Having assigned to each function symbol f_α a free polynomial $P_\alpha(\mathbf{X})$, its fresh scalars $(a_\mathbf{i})$ are introduced into the global scope, and we substitute each call $f_\alpha(\vec{\beta})$ with the body of the polynomial $P_\alpha(\vec{\beta})$. This gives us a constraint satisfying the following grammar, where all free variables are either held in global scope or scoped by a universal quantifier.

$$\langle C \rangle ::= \forall \vec{\beta}.\, C'$$

$$\langle C' \rangle ::= E \mid \bot \mid C' \wedge C' \mid E \Rightarrow C'$$

$$\langle E \rangle ::= \top \mid t = t' \mid t \leq t'$$

Removing universals The last step of processing required to obtain a constraint compatible with solvers is removing the universal variables. Doing so will remove the last of the binders, and in the process. This is where comes in play our refusal to process implications whose hypothesis are term-term equalities involving universal variables.

This is done thanks to the following property of polynomials as vector spaces: the family of monomials $(\mathbf{X}^\mathbf{i})_\mathbf{i}$ over all multi-indices forms a base. Since all our polynomials have degree at most N, we can restrict ourselves to monomials $(\mathbf{X}^\mathbf{i})_{\deg(\mathbf{i}) \leq N}$. Then, given an equality $t = t'$, we transform it into $t - t' = 0$. Now, note that, since our term language is $(0, 1, +, -, *)$, the term $t - t'$ is a polynomial $P_{t-t'}(\mathbf{X}) = \sum_{\deg(\mathbf{i}) \leq N} a_\mathbf{i} \mathbf{X}^\mathbf{i}$ evaluated at some tuple $\vec{\beta}$ of universal variables. Such a polynomial is a vector on the basis $(\mathbf{X})_{\deg(\mathbf{i}) \leq N}$, and therefore, the equation $t - t' = 0$ is equivalent to the conjunction $\wedge_{\deg(\mathbf{i}) \leq N} a_\mathbf{i} = 0$. This last conjunction no longer involves the universal variables. By performing the same transformation over all (in)equalities in the constraints, the universal variables are redundant and can be elided. The only free variables in the constraint are now the scalar parameters introduced when instantiating the functions created during skolemisation.

Note that this method for removing universal variables is not sound in the presence of implications $(t = t') \Rightarrow C$ when the term t or t' contains free universal variables. Indeed, consider the constraint $\forall\beta.\,(\beta^2 = 9) \Rightarrow C$. With this method, the hypotheses becomes $\beta^2 - 9 = 0$, then $1\beta^2 + 0\beta - 9 = 0$, and finally $1 = 0 \wedge 0 = 0 \wedge -9 = 0$, which is absurd. The full constraint would then be equivalent to $\forall\beta.\bot \Rightarrow C$ which is trivially true.

On the other hand, the initial hypothesis is trivially equivalent to $x = 3 \vee x = -3$, making the initial constraint equivalent to $C[3/x] \wedge C[-3/x]$, which is not trivially true. Forbidding term-term equalities involving universal variables in implications, as we did previously, prevents this situation from occurring, and suffices to recover the soundness of the method. A general solution may be out of reach, as such implication may occur under an arbitrary context, which might be relevant to processing it.

While our outright rejection of some hypotheses on grounds of their free variables fails to account for some cases obvious to the human eye (as the previous example shows), it remains quite easy to implement. Furthermore, in the case of AARA, term-term (in)equalities with universals do not appear, as we explained in section 7.6. Indeed, the hypotheses involving universals are always of the form $x = t$ where x is universally bound.

With this caveat in mind, the constraints are now free of quantifiers, and only involve free variables bound in global scope. In this form, the constraint is amendable to integer linear programming. Furthermore, if one or many objectives for optimizations are specified together with the constraint, it forms a classic optimization problem.

Linear programming for closed-form bounds Once the parameter constraint generated by a program with AARA parameters is reduced to a linear programming problem, a closed-form bound on complexity can be given. The objectives of the optimizations are given by the `goal` directive. Recall that it declares a multi-variate polynomial P which we seek to minimize. Just as before, we instantiate this polynomial as some $P(\mathbf{X}) = \sum_{\deg(\mathbf{i}) \leq N} a_{\mathbf{i}} X^{\mathbf{i}}$.

The scalars $a_{\mathbf{i}}$ are the building blocks of the objectives, but cannot serve as objectives themselves. Indeed, one has to set *priorities* between those scalars: highest degree scalars should be smaller, even if it makes a small-degree one larger. But in a multi-variate polynomials, many scalars may have the same degree. Furthermore, there is no *natural* total order on multi-indices that we could use to discriminate between scalars. For example, in the polynomial

$aXY + bX + cY + d$, should the objectives be ordered as (a, b, c, d) or (a, c, b, d) ? For simplicity, we choose to optimize not over the scalars, but over sums of scalars of constant degrees. This means that, in the previous example, we shall optimize for $(a, b + c, d)$, and in general, for the values $(s_N, \ldots, s_0)$, where $s_j = \sum_{\deg(\mathbf{i})=j} a_{\mathbf{i}}$. With the parameters, constraints, and objectives set, an integer programming problem is now well-defined.

Example Let us present this optimization feature with the file `fold.bill` previously studied. The corresponding code is shown in figure 8.6. The first listing show modifications made to the type definitions in the file `fold.bill` (in figure 8.4) that enable optimization. With this definition, the length of the `CoList` computation that consumes a list of length N is now `(P N)`, an unknown quantity. This `P` is defined as a polynomial to be optimized, with degree at most two.

Without making any other changes to the code, Autobill can minimize `P`. To do so, it suffices to call it with the `-skolemsmt` option to print a quantifier-free optimization problem in the SMT-LIB 2 format. The result is shown in the second listing. In lines 1-5, the global parameters are defined, each as an integer equal or larger than zero (some lines were elided for typesetting). Then, the constraint itself is printed, on lines 6-20. Each linear combination of scalars appearing in equalities $t = 0$ asserted to be indeed null. Finally, on lines 21-13, each of the three coefficients of the degree two polynomial $P(X)$ is minimized in turn. Priority is given to the targets of the first `minimize` directives over the last ones. On line 24, the formula for P is printed as to allow the results to be interpreted. The last two lines start the optimization and prints the results.

Those results are shown in the last listing. The first line gives the formula of the polynomial to be minimized, here `P`. The second one expresses that *Z3* found a satisfying assignment, and the last five lines gives the minimal values that were found. All put together, they specify a minimal polynomial of $P(X) = 0X^2 + 1X + 0 = X$, which is indeed correct.

8.8 Closing words

AutoBill is a prototype implementation of our resource analysis method and implementation of parameterized System-**L**. It implements type-checking, evaluation, and normalization of System-**L** programs using the simple type system. Furthermore, AutoBill support parameterized

```
9  decl type P : (nat -> nat)
10 goal P degree 2
11
12 comput Fold (A:+) (B:-) =
13  this.fold_it<N:nat>(List A N, Closure Lin (CoList A B (P N) )).ret(B)
```

(a) Modifications to file `fold.bill`

```
1  (declare-const T_552 Int) (assert (<= 0 T_552))
2  (declare-const T_551 Int) (assert (<= 0 T_551))
3      ... lines elided here ...
4  (declare-const T_541 Int) (assert (<= 0 T_541))
5  (declare-const T_540 Int) (assert (<= 0 T_540))
6  (assert
7   (and
8    (= 0 T_550)
9    (= 0 (+ T_546 (* -1 (+ T_550 1))))
10   (= 0 (+ T_547 (* -1 T_551)))
11   (= 0 (+ T_548 (* -1 T_552)))
12   (= 0 (+ T_543 (* -1 T_550)))
13   (= 0 (+ T_544 (* -1 T_551)))
14   (= 0 (+ T_545 (* -1 T_552)))
15   (= 0 (+ T_550 (* -1 T_540)))
16   (= 0 (+ T_551 (* -1 T_541)))
17   (= 0 (+ T_552 (* -1 T_542)))
18   (= 0 (+ (+ (+ T_550 T_551) T_552) (* -1 (+ T_550 1))))
19   (= 0 (+ (+ T_551 (* T_552 2)) (* -1 T_551)))
20   (= 0 (+ T_552 (* -1 T_552)))))
21 (minimize T_552)
22 (minimize T_551)
23 (minimize T_550)
24 (echo "P(X) = T_550 + T_551 * [X] + T_552 * [X^2]")
25 (check-sat)
26 (get-objectives)
```

(b) Resulting constraint `fold.smt2` (after `autobill -L fold.bill -skolemsmt`)

```
1  P(X) = T_550 + T_551 * [X] + T_552 * [X^2]
2  sat
3  (objectives
4   (T_552 0)
5   (T_551 1)
6   (T_550 0)
7  )
```

(c) Output of optimization process (`z3 -smt2 fold.smt2`)

Figure 8.6: An optimization example in *Autobill*

types with parameters of base sorts. The sort, operations, and relations for parameters can be user-provided, and the ring of natural integers is predefined. Parameterized datatypes and computation types that use those parameters can be defined. Type-checking also support those parameters, even for user-defined sorts. AutoBill then produces, for a given program, a first-order constraint using the signature it defines, exported into the standard *SMT-LIB2* format or as a goal in the *Coq* proof assistant. Normalization of first-order constraints into a compact form is also implemented.

When the input program uses only the predefined natural integer sort for its parameters, the user can also declare a type constructor as a multivariate polynomial to be optimized. In this case, the program's parameter constraint can be translated into a quantifier-free formula over the integer coefficients of the target polynomial by applying a sound simplification algorithm. This simplification relies on the instantiation of unknowns as multivariate polynomials of bounded degree. As the user can provide this degree bound, knowledge by the user of an upper degree bound for their chosen polynomial (i.e. their program's complexity) is enough to obtain solutions in practical cases.

AutoBill is also a collaborative project. It received several contributions from students who implemented a frontend to an ML-style language with call-by-value operational semantics, an imperative fragment for that ML language backed by monadic effects, and a web user-interface in which allows users to interact with an instance of AutoBill ran locally through the browser's Javascript environment or on a distant server.

AutoBill is still a moving target. Some bugs remain in the interaction between linearity and the ML frontend, which causes ML programs compiled to machine code to fail polarity analysis. It would also benefit from a more sophisticated implementation of its type system which would fully cover parameterized System-**L**. This would allow users to define higher-order parameterized data/computation types, that is, type constructors taking parameterized type as arguments. Support of higher-order parameters would also be welcomed, as it would allow polynomials, or even larger classes of functions, to be natively handled in the future. This would in turn unlock the possibility of turning user-provided, un-parameterized programs into parameterized ones for immediate analysis. We plan to implement those features in the future, and some are in already in the works. This would lift a block of the treatment of dependency injection, when a programmer uses a high-order argument in their own functions.

Lastly, deeper integration with SMT solver may allow parameters to be directly instantiated by AutoBill. This may be implemented depending on the success of future experiments in that direction.

Those improvements would provide a streamlined account of parameterized data structures in which the structure of the type is given by its parameterization, such as the list of list $[[], [1], [2, 3], [3, 4, 5], \dots]$, whose inner list at index i is of length i, and contains at position j the integer $i + j$. The type of those integers could be, for example, given by the higher-kinked type $\lambda i.\lambda j.\mathit{Int}(i + j)$.

Through our implementation of parameterized System-**L**, its frontend, and its constraint post-processing, we have demonstrated the feasibility of splitting of resource analysis into a set of steps, that we believe will help the development of future tools. Thanks for the separation of the analyses into a compilation phase to an amendable intermediate representation, a type-checking phase that elaborates parameter information, and a constraint solving phase that re-uses state-of-the-art tools and standard formats, we believe that future additions of new languages, finer notions of dependency expressed in parameter systems, and new solving techniques can incrementally improve the precision and scope of resource analyses for pure and locally-effectful programs. Resource-aware global effects, such as mutable state and exceptions, remain out of scope.

Chapter 9

Related work

The work presented in this manuscript as had as a goal to create original resource analysis systems for λ-calculus based languages. Our approach was to use the formalism of System-**L**, as to factorize the analysis, create interfaces onto which new tooling could be applied, and support more high-level constructs. In this section, we compare our approach to the state of the art by focusing on four specific aspects of resource analysis. First, we focus on the theoretical implications of using small-step v. big-step operational semantics for resource-aware semantics. Then, we shall focus on the advantages brought-on by explicit evaluation contexts for the analysis of typed functional languages. We also will comment the novel design and implementation of a CBPV analyzer along the lines of the AARA method. Finally, we compare the methods chosen by different resource analyzers to compute closed-form bounds from the local information obtained from source code.

9.1 Small-step v. Big-step resource semantics

The resource analysis formalism we introduced in this manuscript is, to our knowledge, the first AARA toolkit implemented over Call-by-push-value semantics of functional programs. This extends the practicality of AARA methods since it doesn't require instrumenting big-step semantics with resources, as must be done with previous techniques.

Recoupable costs In the absence of recoupable resources (i.e. in the case of a *monotonic cost*), knowledge of the relative ordering of ticks is not required to obtain a sound bound. If

two sub-expressions e_1 and e_2 (inducing costs k_1 and k_2 respectively) are evaluated as part of a larger program, it will never matter which is run first: both "e_1 then e_2" and "e_2 then e_1" yield the same cost $k_1 + k_2 = k_2 + k_1$.

On the other hand, if resources are recoupable, the relative ordering between the evaluation of e_1 and e_2 matter: take, for example, suppose that $k_2 = -k_1$ (that is, e_2 frees the resources allocated by e_1). Suppose as well that, before the sequence is evaluated, $K > k_1$ resources are allocated. Then, evaluating "e_1 then e_2" will have a maximum footprint of $K + k_1$, but evaluating "e_2 then e_1" will yield a footprint of merely K. Permuting the relative evaluation order of sub-expressions changed the footprint.

Small-step semantics This ordering-sensitivity influences the amount of work required to make operational semantics resource-aware. When program reduction is expressed in small-steps semantics, and resource information are added step-wise with a resource metric or ticks, the footprints of programs are never ambiguous: individual costs borne by single reduction steps immediately come together into a resource profile, from which a unique, well-defined footprint is computable (see section 2.2). As such, adding local resource information to small-step semantics suffice to define a consistent notion of footprint.

Big-step semantics As opposed to small-step semantics, big-step semantics may not upgrade to resource-aware big-step semantics when ticks are added to programs, requiring some instrumentation. Indeed, consider the following big-step semantics for CBV λ-calculus:

$$\frac{e_1 \downarrow \lambda x.e_3 \qquad e_2 \downarrow V_2 \qquad e_3[V_2/x] \downarrow V_3}{e_1(e_2) \downarrow V_3}$$

When adding resources along the line of RAML, the reduction relation $e \downarrow V$ is annotated with a *watermark* (k, k'), whose intended semantics is: "*if l resources are used at the beginning of reduction, then the maximum footprint during the reduction is $l + k$, and $l + k'$ resources are still used at the end of reduction*". Those watermarks are endowed with an associative concatenation operation written ";" with unit $(0, 0)$, defined as:

$$(k_1, k_1'); (k_2, k_2') = (\max(k_1, k_1' + k_2), k_1' + k_2')$$

Using this annotation scheme, the rule above for function application is enriched as follows, where the watermark (k, k') of the full program is ambiguous.

$$\frac{e_1 \downarrow^{(k_1,k'_1)} \lambda x.e_3 \qquad e_2 \downarrow^{(k_2,k'_2)} V_2 \qquad e_3[V_2/x] \downarrow^{(k_3,k'_3)} V_3}{e_1(e_2) \downarrow^{(k,k')} V_3}$$

indeed, the watermark (k, k') cannot be fully determined from the three watermarks in the premises. While the evaluation $e_3[V_2/x] \downarrow V_3$ must occur after both e_1 and e_2 are reduced to values, which of them is reduced first is irrelevant to the big-step semantics. As such, two valid assignment of (k, k') are possible, namely:

$$(k, k') \stackrel{?}{=} (k_1, k'_1); (k_2, k'_2); (k_3, k'_3)$$
$$(k, k') \stackrel{?}{=} (k_2, k'_2); (k_1, k'_1); (k_3, k'_3)$$

This apparent ambiguity is not a problem when one is concerned with the *structural* safety of programs: no matter which of e_1 or e_2 is ran first, the normal form of $e_1(e_2)$ is unchanged, and whether such normal form even exists in CBV semantics is as well independent of this choice. But as we've seen, the maximum usage of recoupable resource is not a structural artifact of programs: two expressions may reduce to the same value without incurring the same usage.

Consequences for analysis Since resource usage is not a structural invariant, analysis systems based on big-step semantics must enrich those semantics, not only with resource usage (such as the watermarks used in RAML), but also with a *choice* of which sub-expressions are evaluated first. We consider this is evidence of the relative unfitness of big-step semantics for resource analysis when compared to small-step semantics.

This being said, it can be argued that big-steps semantics retain advantages. Indeed, big-step semantics can be produced from small-step ones by defining $e \downarrow V$ iff $e \rightarrow^* V$ and V is a normal-form, thereby eliding intermediate steps. This suggests big-step semantics are simpler than small-step ones, and this simplification can be welcome. Furthermore, the formal issue of having to choose an ordering within premises of big-step rules doesn't pose a problem when one builds an analysis with an implementation in mind. Indeed, such an implementation does provide an ordering of sub-expression evaluation (assuming a single-threaded, non-distributed implementation). Those rebuttals do not come as a surprise, as big-steps semantics have been at the core of contemporary AARA systems such as RAML.

Nevertheless, the simplification induced by using non-ambiguous big-step semantics may come at the cost of changing those semantics away from their conventional forms, such as in the aforementioned case of the CBV λ-calculus. Also, different implementations based on the same formalism may require incompatible choices. For example, RAML is tailored for the evaluation model of OCaml, and provides resource bounds relative to implementation details such as the evaluation order of fields in a record literal.

Lastly, we note that when implementations provide disambiguation to the big-steps semantics, those are in most cases informed by the small-step semantics of some underlying abstract-machine/assembly language/LLVM code/etc... Enriching those small-steps semantics with a cost immediately provides resource-aware semantics. This approach is also chosen for time and memory analysis within the *Costa* termination and resource analyzer[1, 6, 7, 8], which primarily targets bytecode languages *Java* and *C#* through their intermediate representations in the *JVM* and *Common Language Runtime* respectively.

The use of small-steps semantics with ticks, as opposed to manually-enriched big-steps semantics, shows its advantages in the presence of language extensions such as the imperative blocks we introduced in section 5.6. Using the small-step semantics of imperative ML with ticks coming from our references[78, 77], a resource-aware semantics is automatically achieved with ticks, without needing any manual adjustment.

9.2 First-class evaluation contexts

Our presentation of AARA within an abstract machine also enables a simplification of AARA-style type system, bringing them more in line with standard type systems for functional languages and their abstract machines. Indeed, abstract machines such as System-**L** offer first-class, defunctionalized evaluation contexts in the form of stack constructors mimicking the usual data constructors of ADTs.

In order to explain this simplification, it is worth noting two axioms of resource-awareness within AARA systems. First, partial application of functions does not incur a cost due to the body of the function running. That is, all functions of n arguments are values of the form $\lambda x_1 \ldots \lambda x_n.e$, and never of the form $\lambda x_1 \ldots \lambda x_k.e'$ for $k \leq n$ and e a reducible expression eventually producing the rest of the function. The second axiom is a form of quantifier

elimination (which is valid for polynomial, as we've seen in section 8.4), which we formalize in our system as the fact that all constraints $\Theta \vdash C$ are in practice safely reducible to an equation $\Theta' \vdash E$. Those things in mind, let us explain how AARA deals with evaluation context.

RAML evaluation contexts Recall that in AARA, datatypes are enriched with combinatorial *indices* denoting positions at which potential is held (our system is similar, and employs numerical *parameters* as variables to compute total potential). Each datatype constructor is enriched to update those indices when the constructor is introduced and eliminated. This suffices to cover tree-like data structures in purely-functional programming languages with CBV semantics, but only *when paired with specific treatment for functions.* In RAML, this is handled with a type-level stack, which associates to each argument in a call site a type with indices.

This is exemplified by the following resource-aware RAML typing judgment, which serves as a representative of the phenomenon at large. Herein, Σ is an *ordered* list of types denoting the arguments of the current call site, Γ is the usual *unordered* environment of local typed variables, Q is an index for the (Σ, Γ) pair. The, e is an expression of type A A annotated by Q'.

$$\Sigma; \Gamma; Q \vdash e : (A, Q')$$

The presence of the indices Q, Q' is not a significant deviation from mainline type systems which strictly follow the Curry-Howard correspondence: they merely are centralized annotations denoting the amount and location of potential before and after evaluation. Those annotations may be dispatched within the relevant type expressions without changes to the underlying reasoning. The presence of those annotations only at the root of judgments nevertheless provides a simple erasure procedure for the resource-aware type system: removing the indices in judgments reduces the type-system to a simply-typed type λ-calculus with an explicit stack.

This explicit stack is used to carry resource information from call sites to function literals in the presence of partial applications: curried functions can extract potential from their first arguments, have their type refined with those arguments' indices, and be returned without their body being evaluated. This allows the information in the current scope to not be blindly copied into the type produced by the partial application, but only the relevant information from the arguments. This added information refines the index of the partially applied function.

If the resulting function is passed as a higher-order argument to another one, those added annotations are instantiated at the relevant call-site. Therefore, even if the value of this function is dynamically determined, some information can flow from partial call sites to complete ones. Nevertheless, this instrumentation of function literals and call sites does not function when the body of the functional argument is not known in advance, or when the annotation cannot be successfully passed from the function literal to its distant call site.

This scheme is an AARA specificity, which does depart from the well-beaten path of the Curry-Howard correspondence, and may cause problems when such type system is modified to account for new programming constructs, such as polymorphism or potential held in closed-over values.

Abstract machines This departure from more standard type systems can be regularized when using virtual machines. First, the move from λ-calculi to virtual machines that explicitly represent continuations is not a big departure from the valuable Curry-Howard correspondence, as it lines up exactly with a similar move on the logical side: from natural deduction to sequent calculus[15]. While it remains close to standard λ-calculus in that sense, it does resolve the problem of propagating shape and resource information from arguments to call sites, even in the presence of partial applications.

To see this, consider a CBV function type $A_1 \to A_2 \to \cdots \to A_n \to B$, which is translated into CBPV as:

$$\Downarrow A_1 \multimap \Uparrow\Downarrow A_2 \multimap \ldots \multimap \Uparrow\Downarrow A_n \multimap \Uparrow B$$

The presence of the $\Uparrow\Downarrow$ constructors between the arguments guarantees the possibility of partially applying the function, the result simply being a closure over the provided argument that holds the rest of the computation. Assume this function is partially applied with $n-1$ arguments (this is enough for our demonstration). The resulting type is:

$$\Downarrow A_1 \multimap A_2 \multimap \ldots \multimap \Uparrow\Downarrow A_n \multimap \Uparrow B$$

The partial call to the function (let's call it f) proceeds by opening the closure, providing the $n-1$ arguments (let's call them $x_1, \ldots, x_{n-1}$), and forcing the resulting thunk, giving the term:

$$\mu a.\langle f \parallel \mu\Downarrow f'.\langle f' \parallel x_1 \cdot \cdots \cdot x_{n-1} \cdot \Uparrow \cdot a\rangle\rangle : \Downarrow A_n \multimap \Uparrow B$$

When adding AARA parameters to the function f along its CBV semantics, it can be assumed that the function does not perform work when partially applied, i.e. the arguments of the partial application are merely added to the closure it returns. In that case, the quantifiers adjoined to the intermediate closures and thunks are all empty. This yields the following parameterized type before and after the partial application:

$$\begin{array}{llll} \textbf{Before:} & T_{\text{before}} & = & \Downarrow\forall\vec{\alpha}.\, E_{in} \Rightarrow \quad A_1 \to \Uparrow\Downarrow A_2 \to \cdots \to \Uparrow\Downarrow A_n \to \Uparrow\exists\vec{\beta}.\, E_{out} \wedge B \\ \textbf{After:} & T_{\text{after}} & = & \Downarrow\forall\vec{\alpha'}.\, E'_{in} \Rightarrow \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad A_n \to \Uparrow\exists\vec{\beta}.\, E_{out} \wedge B \end{array}$$

The question then becomes, how does one produce the quantifier $\forall\vec{\alpha'}.E'_{in}$ occurring after the partial application? This is immediately resolved by the stack that holds the arguments. Indeed, consider the call site for f as before, now in the parameterized type system[1]:

$$(\Theta \vdash C) \triangleright \Gamma \mid \mu\Downarrow f'.\langle f' \parallel x_1 \cdot \cdots \cdot x_n \cdot \Uparrow \cdot a\rangle : T_{\text{before}} \vdash (a : A_n \to \Uparrow\exists\vec{\beta}.\, E_{out} \wedge B)$$

From this call site, it is possible to produce the desired closure of type T_{after}, simply by setting $\alpha' = \Theta$ and E'_{in} the equation equivalent to C, no further work being required.

In conclusion, the fact that call sites are first-class data structure in System-**L** allows the type system to build the CBV functions calls without any special case being required, even for partial applications. When CBPV semantics are added (which RAML doesn't support), computation and data being on equal footing all the more welcome.

9.3 First implemented CBPV AARA

There was, to our knowledge, no previously developed AARA formalism over CBPV semantics for functional languages. Nevertheless, resource analyses at large are not limited to CBV semantics.

Semantics for AARA The first instance of an AARA formalism, according to Hoffman et al. in their retrospective [34], was due to Martin Hofmann. In this first paper, a linear, simply-typed λ-calculus was endowed with a time and space semantics for which programs were non size-increasing and ran in polynomial time[35, 39, 37]. This complexity result held in

[1]Note that here, we take Θ to be the smallest parameter context that puts all required parameter variables in scope

the purest sense: programs were compiled to the *BC* (Bellantoni-Cook, [14]) class of recursive functions, on which "size" and "time" complexities can be given their standard meaning. This allowed computing the asymptotic complexity of the considered λ-terms using the same scale as used in mainstream complexity theory, but also supporting higher-order functions in the analysed language.

Operational complexity, as opposed to asymptotic complexity in Turing machine, was also an early topic of research for that program. A heap-bounded functional language was also created by Hofmann which compiled to *C* without using dynamic allocation [36]. This gave a qualitative space guarantee (all allocations are static for closed programs), but not a quantitative bound. Bounding of time and space complexity in realistic evaluation models required new methods.

This started with a linear, CBV, first-order functional language with booleans, products, sums, and lists introduced by Hofmann and his student Jost [40]. The heap-space usage of function evaluation was here bounded by a linear expression derived from type-level annotations. We recognize here the core of modern AARA systems: type-level potential annotations induce algebraic inequalities through combinatorial laws, which are then accumulated during type-inference and solved by an off-the-shelf solver.

This was followed three years later with and analysis of a subset of Java, again by the same authors, which required explicit deallocation, but covered essential features for object-oriented programming such as inheritance, coercion down to sub-classes, mutable update, and aliasing. Jost continued the work of extending the features supported by AARA, notably authoring (in collaboration with others) support for algebraic data types and recursion[44], for higher-order functions and polymorphism[43], and for call-by-need semantics in 2017[45]. Mutable arrays and reference with null potential were added by Litchman & Hoffmann[56], and ideal garbage collection by Niu & Hoffmann in [65]. This is nevertheless far from exhaustive, and we recommend the interested reader to peruse one of the review paper dedicated to the technique[50, 38, 34].

The main field of application of AARA remain anchored in the typed-functional family of languages. This should not be unexpected, as AARA consists of a type system enriched with reasoning capabilities sound with regard to rewriting-based semantics, in line with the Curry-Howard correspondence. This does not mean that programming paradigms not falling

under that description lack resource analyses, or that those analyses cannot be inspiration for resource analysis of functional languages. Let us quickly mention amongst those methods based on abstract interpretation and those based on the extraction of recurrence relations.

Typing and constraint solving The main principles of our type system are inspired by Dal Lago's work exploiting the relation between type inference and constraint solving to derive complexity results. In collaboration with Petit, PCF was endowed with a linear, dependent type system, in which the number of substitutions required to evaluate a program was embedded in typing judgments[21]. Their main result is an inference procedure which reduces type inference for this high-order, typed language to a constraint solving problem in first-order arithmetic. Later works generalized this approach by parameterizing the language of constraints and the solver attached to them, making the system more usable and paving the road for the use of practical solving procedures[20]. Finally, Rajani et al. introduced abstract notions of resources and cost through the use of a monad/comonad pair graded by the amount of resources held/spent[69]. We consider this reduction in order is a key idea to building resource analyzers for typed functional programs, as it enables a welcome separation of concerns between the specifications of resource semantics, size-and-cost-aware typing, and constraint solving to obtain closed-form bounds.

AutoBill AutoBill was developed as a static analysis framework for statically-typed functional programs, which bounds it range of application. But as we'll see, it extends the range of applications of previous AARA analyzers by integrating core ideas of dℓPCFand λ_{amor}, and recovering parts the modular nature of CiaoPP, and the symbolic reasoning of Costa.

Call-by-push-value semantics strictly extend the call-by-value semantics which were the main point of focus of previous AARA work. In that sense, while CBV semantics only allow for the "function" computation, our thunks, closures, and stack constructors open a whole universe of computation types with varying calling conventions. But this point of view is reductive, as linear call-by-push-value enable an expressive interleaving between manipulation of data through positive types, and specification of computations through negative types.

CBPV semantics have a close relation to the (co)monadic semantics of dℓPCFand λ_{amor}. Indeed, CBPV is built on an *adjunction* between positive and negative types, and all such adjunctions define a monad/comonad pair. This process isn't one-to-one, as different adjunctions can

derive the same (co)monad. In the field of resource analysis, this has large effects on resource semantics: our system involves explicit resources, whose usage is well-defined at each step of computation. Resources cannot enjoy the same status in (co)monadic semantics, they only exists as cost or potential associated to a value or computation. To our knowledge, this work constitutes the first AARA resource semantics built on top of this strong basis. AutoBill incorporates both the high-level aspect of state-of-the-art AARA analyses (with resources held as potential within scope), but also an operational, small-step, low-level formalism (where resources are first-class and changes in state across time can be accounted for as atomic transitions).

Finally, CBPV semantics enable encapsulation of data, computation, and specification akin to the object-level specification of *Costa*. Indeed, some object-oriented paradigms are expressible, namely (immutable) objects and inheritance of interfaces. This can be achieved by defining a type T satisfying the following equation:

$$T \simeq \Downarrow\big((A_1 \multimap \Uparrow B_1 \otimes T) \,\&\, \ldots \,\&\, (A_n \multimap \Uparrow B_n \otimes T)\big)$$

This type then represents an object with n methods, each taking an argument A_i and returning a B_i together with a modified object. The closure holds the reference to the instance variables of the object. This encapsulation also functions at the resource-level. Consider now the same equation, now enriched with parameters:

$$T(\vec{\alpha}) \simeq \exists\vec{\beta}.C \wedge \Downarrow \Big(\Big(\forall\vec{\gamma_1}.C_1' \Rightarrow A_1 \multimap \Uparrow\exists\vec{\delta_1}.C_1'' \wedge B_1 \otimes T(\overrightarrow{\alpha_1'})\Big) \;\&\, \ldots \\ \ldots \,\&\, \Big((\forall\vec{\gamma_n}.C_n' \Rightarrow A_n \multimap \Uparrow\exists\vec{\delta_n}.C_n'' \wedge B_n \otimes T(\overrightarrow{\alpha_n'})\Big)\Big)$$

In this parameterization of the object type T as $T(\vec{\alpha})$, its behavior is specified by *public* parameters $\vec{\alpha}$, and *private* parameters $\vec{\beta}$ who are only partially specified by C, which represent the encapsulated state admitting an invariant C. Then, each method specifies the parameters of its argument A_i with the quantifier $\forall\vec{\gamma_i}.C_i'$, and returns some result B_i whose parameters $\vec{\delta_i}$ are introduced by $\exists\vec{\delta_i}.C_i''$, as do normal functions. Finally, the modified object $T(\overrightarrow{\alpha_i'})$ is returned, with a new parameterization that still satisfies the invariant. As such, this type can indeed represent object-level specification of resources and sizes, while guaranteeing safe operation through encapsulation in an abstract type. The type T only specifies an interface, which can be used as part of another object extending its interface, or shimmed to implement only a subset of its features.

By using linear call-by-push-value semantics, and implementing a types-and-constraints formalism for resource analysis, AutoBill marries a low-level operational semantics, which allows for fine ordering of evaluation steps and backs resource bounds, together with a high-level, component-based interface with behavioral specification.

9.4 Constraint programming

The final step of our resource analysis is a constraint solving step which transforms an implicit complexity bound into a closed-form expression. In currently implemented AutoBill, this is a two steps process. First, the first-order constraint over polynomials with integer coefficients is transformed into a linear integer optimization problem those coefficient, and the second is a call to a readily available solver to instantiate them. While the later step is standard in AARA implementation, the use of a syntactic constraint in the former is a departure from RAML. We now compare this approach and its trade-offs with other techniques.

9.4.1 Constraint programming for AARA

AARA systems emphasize type systems and related techniques as opposed to constraint programming. A goal of this section is to relate AARA techniques and constraint solving. To begin with, let us review some AARA formalisms.

Linear AARA The index system associates to each variable in scope not only a type, but a sum of *potentials* whose *locations* are given by an index. The location the index points to should be understood as one or many nodes in a tree-shaped data structure. This gives a bound to the range of application of AARA: index systems only functions on algebraic datatypes, and fail on cyclic structures and cyclic heaps. But the syntax of indices also determines the reasoning power of an AARA system.

For example, *linear AARA* does not place potential deep within a data structures. Namely, the types and indices of linear AARA are (isomorphic to the following grammar, where B is a base type)[2]:

[2]As to accommodate the simultaneous description of many AARA index systems, changes have been made to some of them to unify their notation and representations.

Types			*Indices*			
T	$::=$	B	I_T	$::=$	$*$	*(base types)*
	$\mid$	$(T_1, \ldots, T_n)$		$\mid$	$\pi_i(I_{T_i})$	*(tuples)*
	$\mid$	$T_1 + \cdots + T_n$		$\mid$	$\iota_i(I_{T_i})$	*(sum types)*
	$\mid$	$L(T)$		$\mid$	$L(I_T)$	*(lists)*

Each I_T defines a set of positions within the tree realized by a value of type T. The index $*$ denotes the root node, the projection ($\pi_i(I)$) denotes positions I of the i^{th} component of a tuple, the injection $\iota_i(I)$ denotes positions I in the i^{th} argument of a sum type (and the empty set if the value uses another constructor), and $L(I)$ denotes the positions I for all elements of a list.

When introducing or eliminating a constructor, the indices of its arguments are modified, and their coefficient are constrained as to preserve the overall potential in programs. For example, given a list $l : (L(T), p \cdot L(*) + q \cdot *)$, each *cons* node is assigned p potential and the *nil* node q potential. Then, adding another another element will cost p resources, and pattern-matching on l will release either p resources if the list isn't empty, or q if it is. The coefficients $p, q, \ldots$ are not constants, but variables bound in constraints.

The form those constraints take is determined by the need to express conservation of potential (or its relaxation, *non-increasing potential*) within typing rules. Since conservation of potential for introduction and elimination only involves constant amounts of potentials, which means the preserved quantity is a sum of the unknown $p, q, \ldots$, i.e. a linear combination. Likewise, AARA functions types$(A, Q_A) \xrightarrow{p/q} (B, Q_B)$ state the conservation of potential between the $p \cdot * + Q_A$ potential when entering a call, and the $q \cdot * + Q_B$ potential as it returns, once again a linear combination. Therefore, the only constraints generated on the coefficients are linear. If one types a closed program $\overset{p}{\underset{q}{\vdash}} e : (T, Q_T)$ and seeks to minimize the starting potential p, it suffices to collect linear (in)equations $\vec{E}$ during typing, and use a solver for *linear optimization* to find a small $p \in \mathbb{Q}_+$ such that E. The same procedure can be used for functions. For example, consider the following function taking lists of lists as arguments:

$$\vdash e : (L(L(B)),\ q_1 \cdot * + q_2 \cdot L(*) + q_3 \cdot L(L(*))) \xrightarrow{p/q} B$$

Once the coefficients are solved for, the cost of the program can be estimated by counting the number of locations that each index points to. The base index $*$ accounts for a unit cost, $L(*)$ for the length of the outer list, and $L(L(*))$ for the sum of the lengths of each of the

inner list. Should we assume a outer length of n, and inner lengths of $(m_i)_{i<n}$, the total cost is $q_1 + q_2 n + q_3(\Sigma_i m_i)$.

Linear AARA is not suitable to derive general polynomial bounds. For example, [34] determines that it cannot derive the complexity of $\lambda x, y.quicksort(append(x, y))$ as $O((|x| + |y|)^2)$. Therefore, a finer index system is needed to account of cost involving polynomials, but also logarithms, exponentials, etc.

Multivariate polynomial AARA The linear potential invariants created by the typing rules of linear AARA can be replaced to derive polynomial bounds for a wider class of algorithms, while keeping intact the linear optimization that obtains fine bounds. Namely, indices no longer point at the locations of nodes in structures, but in the locations of *patterns* within those structures consisting of many nodes. This is the system described in section 2.5.

The multivariate polynomials used in this case are not only closed under addition and n-ary composition, but also under multiplication. This makes the representation of indices non-trivial: given two linear combinations of monomials, their product might be a linear combination with a quadratic number of monomials. As annotations are multiplied, this can quickly get out of hand. To keep the size of the data structures involved in the analysis manageable, RAML implements a limitation in the degree of polynomials, nullifying higher-degree terms when they occur. This bound is set by the user of the tool.

A similar issue is at the heart of the support of linear structures (linked list & arrays) in RAML. As to simplify the expressions involved in the analysis of linear structures, the RAML contributors observe in[34] that:

> *"Typical polynomial computations operating on a list $v = [a_1, \ldots, a_n]$ consist of operations that are executed for every k-tuple $(a_{i_j})_{j \leq k}$ with $1 \leq i_1 < \cdots < i_k \leq n$."*

This motivates the use of a specific base of polynomials, namely the binomial coefficients. A computation that performs a operation of polynomial cost $C(\vec{x})$ on all k-tuples of a list of length $l(\vec{x})$ will then have a total cost $C(\vec{x})\binom{l(\vec{x})}{k}$.

Thanks to this, it is possible to unify the potential before and after the introduction/elimination rules without solving generating polynomial constraints. Indeed, when introducing or eliminating a constructor C for an algebraic datatype, k-tuples inside the arguments of the

constructors are related to the $(k+1)$-tuples sitting inside the larger value that start with C. The combinatorics used to count those tuple are naturally described as binomials. To apply this, consider a constructor K for an ADT T defined by:

$$y : (A, Q), \overrightarrow{x_i : (B, Q_i)} \vdash K(y, \vec{x_i}) : (B, Q')$$

Each annotation Q_i and Q assigns to each index of B a rational variable. Furthermore, those indices are sequences $[(K_1, I_1), \dots (K_p, I_p)]$ denoting the number of p-tuples of nodes in a value of type T in which the i^{th} term as root constructor K_i.

To easily compute the coefficient of Q' at an arbitrary index $I = [(K_1, I_1), \dots (K_p, I_p)]$, check if K_1 is the constructor is the used constructor K. If it isn't, just add together the contribution of each Q_i at I. Otherwise, let q_i the coefficient of Q_i at $I' = [(K_2, I_2), \dots (K_p, I_p)]$, and q (resp. q') the coefficient of Q' (resp. I'). The $(k-1)$-tuples at I' in the argument x_i remain at I' in $K(y, \vec{x_i})$, but also make a k-tuple at I. As such, their coefficients are described by the constraint $q + p' = \sigma_i q_i$. This is the basis of the *potential unfolding* transform in AARA, which generates the constraints generated when introducing and eliminating constructors.

Compared to polynomials expressed using the standard basis, the unfolding generates a significantly reduced number of linear constraints, and does not require conversion to-and-from the binomial basis to exploit combinatorial properties for resource analysis. As a result, unification of resource annotations during type inference can proceed while only generating linear constraints, even with polynomial potential.

Extension to index systems Since the system of indices determines the kind of algorithms and complexities an AARA system can handle, a question arises: can it be extended to support more than uniform iterations on algebraic datatypes ? Further work has indeed provided such extensions. While they have yet to be implemented, and may therefore involve non-trivial practical difficulties, they do extend the range of application of AARA in theory. Three main examples are, in our opinion, quite relevant: exponential complexity, logarithmic complexity, and general recursion over ADTs.

Exponential indices, introduced in [46] are built in the exact same way as polynomial indices, but changing their semantics from binomial numbers to *Stirling's numbers of the second kind*. Those numbers $\left\{ {n \atop k} \right\}$ (for $n, k \in \mathbb{N}$ and $k \leq n$) enumerate the number may to partition a

n-element set into k partitions. They can be explicitly computed as:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i (k-i)^n \binom{k}{i}$$

Note that $\left\{ {n \atop k} \right\} \in \Theta(k+1)^n$, and that therefore Stirling's numbers of the second kind are suitable to encode a base of exponential functions[3]. Furthermore, just like the binomials before them, they are subject to simple linear relations, of which:

$$\left\{ {n+1 \atop k+1} \right\} = (k+1) \left\{ {n \atop k+1} \right\} + \left\{ {n \atop k} \right\}.$$

This expression can naturally relate the cost of a computation on a patterns of size $k+1$ in a structure of $n+1$ elements, in terms of the cost of the same computation on a smaller structure done over patterns of size k and $k+1$.

This has applications for the analysis of NP-complete problems, such as the SubsetSum problem, which seeks to know, given a list of integers and a fixed target, whether some sub-list sums to the target. For example, consider the following *OCaml* algorithm that solves SubsetSum:

```
let subset_sum (l: int list) (target : int) : bool =
  function
  | [] -> target = 0
  | h :: tl -> (subset_sum tl target) || (subset_sum tl (target - hd))
```

AARA can bound the number of calls to $=$ and $||$ as $1 + 3\left\{ {n+1 \atop 2} \right\} = 3 \cdot 2^n - 2$, which in this case is tight. Furthermore, the exponential and polynomial indices can be combined using as a basis for potential the functions $\varphi(n,k,m) = \binom{n}{k} \left\{ {n+1 \atop m+1} \right\}$, which satisfy sufficiently good combinatorial properties: φ marginalizes to binomial and Stirling numbers, and potential can be transferred to-and-from polynomial and exponential indices by way of using the identity:

$$\left\{ {n+1 \atop 2} \right\} = 2^n - 1 = \sum_{i=1}^{n} \binom{n}{i}.$$

Since exponentials and polynomials can be combined, one can wonder if polynomials and logarithms can receive the same treatment. Hoffmann, Leutgeb, et al. introduced in [41] the

[3] As in, a base of their asymptotic behavior as arguments go to infinity.

potential functions *rank* and p for a binary tree datatype:

$$\begin{aligned} \mathit{rank}(\texttt{Leaf}()) &= 1 \\ \mathit{rank}(\texttt{Node}(l, x, r)) &= \mathit{rank}(l) + \mathit{rank}(r) + \log|l| + \log|r| \\ p_{a,b}(t) &= \log(a \cdot |t| + b) \end{aligned}$$

The corresponding indices are then $*$ for *rank*, and $(a, b) \in \mathbb{N}^2$ for $p_{a,b}$. This allows, for example, to derive a tight logarithmic bound for the lookup operation on *splay trees*. Splay trees are an amortized data structure featuring $O(\log n)$ insertion, lookup, and deletion of elements in the worst case scenario, but provides fast access to the most-recently accessed elements ([74, 64]). They are built from a constant-cost function called *splay*, which recursively rotates the tree as to bring a target element to the root (or the closest element smaller than the target). Using those indices, it is possible to derive a cost of $3\log|t| + 1$ iterations to splay a tree t. Note that in this case, finding normal form for cost is a non-trivial matter, as simplifying the cost may worsen the bound. Indeed, since $\log(a + b) \leq \log a + \log b$, finding a closed form as a polynomial of logarithms from an expression with nested logarithms and sums is a non-trivial matter.

Finally, more general indices have been developed to exploit the tree-like nature of algebraic data types, allowing for much finer bounds than the multi-variate indices we presented earlier[29]. The latter only support enumeration of *sequences* of nodes in structures, but some bounds can only be derived when enumerating arbitrary sub-trees instead. To this end, an index system for *regular recursive types* is defined. Recall that those regular recursive types are either a unit or base type, product or sum type, or a fixpoint $\mu\alpha.t$, where t is a regular recursive type and α a type variable in scope in t. For example, linked lists can be defined a the type $l(a) = \mu t.1 + a \times t$, binary trees as $\mu t.1 + t \times t$, and arbitrary trees (where each node has a list of children), as $\mu t.1 + (\mu l.1 + t \times l)$. This is an extension of the multi-variate index system of RAML. In now includes an index $\mathbf{fold}I$ for fixpoint types $\mu\alpha.t$, which represents an index I being placed at all occurrences of α in t. Iterating this constructs allows indices to "dig" inside tree structures, representing, for example, all pairs of ancestor and descendant nodes in a tree of arbitrary shape. Practical applications involve traversal and accumulations on arbitrary trees. The authors of [29] provide as example a function that, given a filesystem, returns the list of all (*file*, *folder*) pairs where *file* can be accessed by descending from *folder*, whose potential annotation detects correctly the quadratic time cost.

Conclusion As we showed in this whirlwind tour of AARA index system, a main characteristic of AARA index system is their lack of direct constraint manipulations. Apart from generating linear equations, no "solving" actually occurs. Instead, indices are directly "computed". But this apparent lack of constraint solving in AARA implementation hides a deeper insight regarding those indices: they are compact, combinatorial encodings of functions in $\mathbb{N}^k \to \mathbb{N}$ tailored to the operations of folding/unfolding that occur in an AARA analysis.

More specifically, for a fixed type T, indices represent functions $p \in P_T$ from values of that type to integers that are partially ordered, can be linearly combined with coefficients in $\mathbb{N}$, include constants, have a folding relation $p \triangleleft p'$ between P_T and $P_{A\times T^k}$, and an unfolding relation $p' \triangleright p$ in the other direction, all such that $p \triangleleft p'$ and $p' \triangleleft p''$ implies $p \geq p''$ (folding then unfolding decreases potential). Instead of working in a logical theory for those sets of functions P_T, RAML and other AARA systems directly work with indices, in which the folding and unfolding are implemented as functions which turn the two relations deterministic, as would a proof assistant executing a fixed strategy to find a witness of $\exists p'.p \triangleleft p'$. As such, RAML can be seen as an ad-hoc solver for some theory of functions, which relies and standard linear programming toolkits for reasoning on $\mathbb{Q}$ to find witnesses for coefficients.

It is therefore notable that this solving-adjacent technique is not implemented using standard tooling. Given that different index system can be implemented on top of the same language, representing the potential functions of AARA abstractly in the type system and delegating their resolution to dedicated tools would simply implementations. It would furthermore not impede the use of ad-hoc solvers with current indices systems.

9.4.2 Constraint solving & *Costa*

The *Costa* analyser for *JVM* bytecode proceeds through this (simplified) pipeline. First, the control-flow graph of a program is determined, and its loops are extracted into distinct subgraphs. Then, this control flow graph is compiled into a set of Horn clauses relating the value of variables in and out of the basic blocks of the graph, which is called a *Rule-Based Representation*, and is specifically implemented as *Cioa Prolog* programs. Auxiliary steps promote the stack offsets of the JVM into variables and transform block into their *single static assignment* form.

From then on, *Costa* proceeds by steps of abstract interpretation, first by nullity and sign analysis, and then by a more complex size analysis. This is provided by the *Ciao Prolog* toolkit for abstract interpretation. This, in our opinion, constitutes another analysis phase related to constraint solving.

Once those analysis phases are done, a system of mutually-recursive *recurrence relations* is derived from the rule-based representation, each of them guarded by a predicate on sizes and iteration counts. The last step of the analysis is a derivation of a closed-form upper bound for the total cost of the program from those recurrence relations. Finding this bound is done through a bespoke solving procedure which forms the second constraint solving step of the analysis.

It is important to note that the resource analyses built upon CiaoPP are but a part of a larger analysis/optimization framework, itself made of a pipeline of analyses and transformations for logic programs. We wish therefore to highlight the possibility of combining abstract interpretation and other techniques for resource analysis.

Recurrence relations Costa factors its cost analysis through the use of recurrence relations (RR)[4]: once those are generated from a program, the problem of cost bounding reduces to one of bounding a particular function in the RR. Recall that those recurrence relations are a finite family of mutually recursive equations guarded by some relations obtained through size analysis. We can represent such a family for relations/multi-valued functions $(f_i)_i \subset \mathcal{P}(\mathbb{Z}^n \times \mathbb{R})$ as a list of *Horn clauses* C_j:

$$C_j := \forall \vec{x}, \vec{y}. R_{j,1}(\vec{x}, \vec{y}) \wedge \ldots R_{j,n}(\vec{x}, \vec{y}) \Rightarrow f_{i_j}(\vec{x}) = e_j[\vec{x}, \vec{y}, \vec{f}]$$

Those relations exhibit non-trivial behavior which are worth mentioning. First, the guards in each C_j need not be mutually-exclusive, even when they both guard an equation for the same function. In this case, the overlapping relations must be satisfied simultaneously. Second, in each C_j, the value of $f_{i_j}(\vec{x})$ is defined in terms variables $\vec{x}$ and $\vec{y}$, but the later need not have a single value: they may only be partially specified by the guard. Lastly, the well-foundedness of the recurrence is not obvious, and in fact not assumed. A computation of a value for a function may involve a non-trivial sequence of recursive calls, and may not even terminate.

[4]The authors of Costa differentiate between non-deterministic, multi-ary RR, which they call "cost relations", reserving the term RR for deterministic, one-argument recurrence relations. We believe the term RR is more explicit even in the former case.

As such, solvers for those RR also double as explicit termination analyzers: if a closed-form is found, then the recurrence is well-founded, and in the case of time cost, the evaluation of a program is guaranteed to terminate.

Solving CR The solving of the recurrence relation then corresponds to finding an expression e_* giving a closed-form upper bound of a previously selected function f_* in the relation, that is, to satisfy the constraint:

$$\forall \vec{f}.\, C_1 \wedge \cdots \wedge C_n \Rightarrow (\forall \vec{x}. f_*(\vec{x}) \leq e_*[\vec{x}])$$

This solving step is handled by a separate component of the resource analyzer: the *Practical Upper Bounds Solver (PUBS)*, described in [5]. PUBS is independent of the Costa analyzers, and may be reused in other analyses or for purposes outside of resource analysis. PUBS handles linear, logarithmic, and exponential complexities simultaneously. It proceeds by first choosing functions that, given arguments $\vec{x}$, bounds the number of internal nodes and leaves of the call tree generated by the evaluation of $f_*(\vec{x})$. This is done through the technique of *ranking functions*, which assigns to each $\vec{x}$ an element in a well-founded partial order. Then, for each cycle occurring in the RR, a linear invariant is determined over its arguments through abstract interpretation in the polyhedral domain. Unfolding relations to obtain only direct recursions simplifies the problem, which the authors claim is feasible in RR occurring in the wild.

Finally, those invariants are then used to bound the immediate (i.e. non-recursive) part of each f_i by individually bounding each "basic" linear expression occurring at the leaf of its syntax tree. Note that the language of expressions in RR is designed as to guarantee that this is a sound way of finding a global bound. This, together with the bounds on the call trees, suffices to generate a closed form bound. In practice, bounding the cost of each *level* of the call tree (nodes at constant depth from the root) is more precise than bounding individual nodes separately. This helps when *divide-and-conquer* algorithms are used in source programs.

Conclusion *Costa*, as well as other analyses not listed here (such as [57] and [72]) combine abstract interpretation and recurrence relation solving to obtain resource bounds. This approach shows the usefulness of other static analyses, such as size analysis and invariant generation, to resource analysis. Indeed, this can enrich the intermediate representation of

source programs with essential information which may be difficult to obtain in parallel to a resource bound. For example, understanding the memory footprint of a data structure implies inferring its size, and likewise, understanding the time complexity of an algorithm relies on understanding its invariants.

Here, the reusable tools (abstract interpreter, size and invariant analyses) are combined with an ad-hoc solver (PUBS for recurrence relations). Compared to AARA, this approach is more modular, and provides more explainable bounds and intermediate results compared to AARA indices. Nevertheless, it should be mentioned that the source languages targeted by Costa and Ciao in general are first-order languages in the imperative family (Java bytecode and LLVM-IR) and logical programming (a subset of Prolog). On the other hand, AARA systems scale better in the presence of higher order programs: while the cost of high-order argument is not taken into account, its effect on the indices are inferable and are transferred to the call site, leading to finer bounds.

9.4.3 *Solving Modulo Theory* and program synthesis

Liquid resource types AutoBill is not novel in its use of a combined SMT solver and type system to derive resource bounds for functional programs. Indeed, *Liquid Resource Types*[48] may be used for the same purposes. Those are based on *Liquid Types*[71, 79] a type system which combines refinement types with SMT solvers to prove non-trivial properties of programs. *Liquid Haskell*[5] is an implementation of such a system, in which annotations combining types and predicates can be used to create types of values satisfying some invariant. For example, Liquid Haskell can represent non-empty lists as `{v:[a] | 0 < len v}`, where `len` is the usual list length function, as opposed to a type-level primitive. This is a form of dependent typing with implicit witnesses, that is without manipulation of values with proposition types. Nevertheless, types may require naming variables in predicate, as in a function type in which the name of the argument is used in the return type. Furthermore, explicit proofs may be needed to guide the solver through some reasoning. Liquid Haskell makes use of an SMT solver to simplify typechecking/proving, and allows lightweight metric annotations to automate the proofs of terminations of recursive computations.

[5] https://ucsd-progsys.github.io/liquidhaskell/

Liquid resource types extend the syntax of liquid types only by letting types bear an annotation of type `Int` representing potential that they must hold, written a^p for a type a and potential p. For example, the function `insert` on sorted linked lists can be given any of the three types annotations below.

$$\begin{array}{lrcccc} \texttt{insert}: & a & \longrightarrow & [a] & \longrightarrow & [a] \\ \texttt{insert}: & a & \longrightarrow & [a^1] & \longrightarrow & [a] \\ \texttt{insert}: & (x:a) & \longrightarrow & [a^{\texttt{ite}(\lambda v.v<x,1,0)}] & \longrightarrow & [a] \end{array}$$

In the second type annotation, the a^1 annotation specifies that each element of the list argument must hold at least one unit of potential which is no longer present in the output, giving a linear cost estimate to the insertion. In the third one on the other hand, the annotation $\texttt{ite}(\lambda v.v < x, 1, 0)$ uses a conditional `ite` to specify that a generic elements v holds a potential of 1 when it is smaller that x and 0 otherwise. Still still gives a linear bound in the general case, but allows for finer bounding when the type a is instantiated with a refined type which determines the condition: this is a value-dependent bound on the function.

The usual potential annotations may be, in a sense, recovered with liquid resource types. For example, a type of lists carrying potential quadratic in its length can be created by specifying that the `cons` constructor takes an element $x : a$ and a tail $t : [a^1]$ of elements holding one more unit of potential. This means that lists hold elements of type a^0, then a^1, a^2, a^3,etc. This is quite similar to the AARA index $1 \cdot [(\texttt{cons}, *), (\texttt{cons}, *)]$ which specifies that each element one unit of potential per element after of it. But Liquid resource types also extend this inductive potential scheme with *abstract* potential, that is, by replacing the unit cost increment in the tail (the 1 in the type $[a^1]$) by a generic function q taking the head and tail of the list as arguments. Type-checking will then use an external solver to instantiate a value of q that preserves the potential rule. This will require a solver supporting *Second-Order Conditional Linear Arithmetic*, which is done using the *ReSyn* solver.

Constraints and program synthesis The use of second-order constraints is only required in the case of dependent potential annotations. When potential annotations are all constant, typechecking reduces to mere first-order conditional linear arithmetic, for which SMT solvers can be used. Namely, one can instantiate function templates whose "holes" are integers variables, which reduces the problem to a first-order one. Complexities other than polynomials can also be verified through a "savant" choice of data structures. For example, the exponential

complexity of a *SubsetSum* algorithm can be verified by using a binary tree whose leaves correspond to all possible selections of elements in the sum.

In the high-order case, the use of more involved techniques are required to obtain bounds through *program synthesis* techniques. *ReSyn* uses *counterexample-guided inductive synthesis*[12] to iteratively generate and test syntax trees that could stand for potential functions. This proceeds starting from a dummy candidate, then uses an SMT solver on the first-order problem resulting from the candidate. If the solver answers favorably, synthesis as succeeded. Otherwise, the solver produces a counter-example. A new candidate is then generated that satisfies this counter-example, and the process begins anew. All counter-examples are kept in memory, and all new candidates must satisfy them simultaneously. The choice of candidates is dependent on the underlying theory the problem is stated in, and randomized or heuristic methods may be required.

Conclusion Liquid resource types combine general-purpose *liquid types* backed by an SMT solver with program synthesis techniques to instantiate AARA index coefficients. This is in turn used to verify cost bounds for functional programs where full inference fails, for example in the case where value-dependent potential annotations are needed. In our eyes, this constitutes a combination of the approaches used in AARA and Costa: costs are express using potential annotations on types, whose closed-form is synthesized rather than set on stone through the use of base index system as RAML does.

9.4.4 AutoBill

Autobill doesn't implement a fixed resource analysis procedure, but instead can be used to encode many of them to obtain a first-order constraint, which in the case of AARA can be processed further to eliminate quantifiers. Furthermore, Autobill's intermediate representation of programs using System-**L** can be used to implement many different programming languages. By separating analysis into a compilation into a linear, polarized, continuation-passing machine, a pass of size-and-resource aware typing, and a constraint solving phase, Autobill splits the resource analysis problem into smaller, more manageable pieces. This opens the door to several independent avenues to improving resource analysis systems in the future.

By allowing CBV and CBN semantics to co-exist, Autobill's moves away from the particular semantics of a given programming language, unlike RAML and Liquid Haskell. As we have shown in this manuscript, there is no need to tightly couple a source language and a constraint solving algorithm to obtain bounds, as Costa has done with a separate solver for its cost relations. On the other hand, Costa is tailored for processing imperative programming through reasoning on pointer arithmetic. System-**L** (as we used it) is not a low-level program representation, and allows for treating high-order values and computations without having to first compile them away. We believe this has advantages in the support of pure functional programming, as it stays closer to the algorithms intended by the programmer, and supports size-and-resources annotations, which said programmer could not, in general, be able to assign to low-level bytecode.

From this base, Autobill produces not a closed-form complexity bound as RAML would, but an implicit complexity bound in the form of a first-order constraint with a distinguished function symbol standing for a complexity function. This is once again closer to the methods Costa and Liquid Haskell use to elaborate closed-form bounds and type-level programs. This is a further decomposition which we believe is an improvement over more tightly-coupled methods of bound elaboration. Indeed, given a fixed language and resource metric, the kind of constraint reasoning required to obtain a bound can vary widely depending on the algorithm at hand. Adding support for the relevant reasoning should, ideally, have as little impact on the parts of the analysis dedicated to the operational and resource-aware semantics of the source language.

The constraint language used in this situation is of critical importance. Indeed, it determines which external tools can be used to validate or elaborate an implicit resource bound. Costa uses a purpose built solver for its resource recurrences, Liquid Haskell either an SMT solver or the ReSyn second-order-arithmetic solver, and we interface with the Z3 solver. This choice in turn dictates whether some property of the program's complexity can be asserted. For example, Costa can export closed-form logarithmic bounds, while RAML and ReSyn fail to derive $O(n \log n)$ bounds for implementation of merge sort. In this case, for example, the Liquid Resource Type method implements merge sort using a balanced *merge tree* containing n leaves, and therefore $O(n \log n)$ nodes. The fact that the merge tree is indeed balanced requires manual proof, as does the fact that the modified algorithm matches the original merge sort implementation. As such, the choice of solver influences not only which kind of complexity

can a resource analyzer reason about, but also how automated it can be. Our choice of solver (the Z3 SMT solver) was then informed by its support of full first-order constraints for verification, its support of multi-objective optimization for quantifier-free constraint, its use in industry settings, and support of the SMT-LIB file standard.

9.5 Conclusion

We claim that the decomposition of resource analyzers into, first, a pre-processing phase which explicits operational semantics, then the elaboration of a constraint, and finally a resolution stage is beneficial. It enables new possibilities of improvements through collaborations between the different fields of computer science involved (semantics, type system implementations, and constraint solving). Nevertheless, the difficulties inherent in automated resource analyses remain. It is still, in general, Turning-complete to compute a resource bound for a program, and more pragmatically, to derive such a bound for a program found in the wild.

Where the latter limitations are met, compromises must be made between the programmer and the resource analysis system. For example, algorithms which are "correct by construction", such as a tree-backed merge sort, are easier to bound since important invariants about the algorithm are tracked in the additional data structures. This implies that, *in fine*, more information in given to the analyzer by the programmer, which may not be a worthy tradeoff in practice. A more reasonable expectation is to ask the programmer to annotate problematic implementation with an annotation to be verified, or even taken at face value, by the analyzer. In the struggle to find acceptable compromises to ensure bounds are found, we find that our questions now meet the concerns of software engineering with static typing and the design of semi-automated proof assistants, which features elaboration of implicit types and value and the guiding of the assistant through a large search space through (ideally minimal) annotations.

Chapter 10

Conclusion

In this manuscript, we asked the general question of how to predict the resource footprint of a program, considering that a tension was evident between the extent and precision of those predictions and the compromises programmers would have to make to obtain them. Given the diversity of systems created to perform resource analysis, each with its own specialization and strong points, we considered that it was not judicious to add another redundant tool to the toolbox, but to design a novel tool capable of iteratively implement many analyses on many different languages.

This required not reusing a specific source language, already subject to its own compromises and design choices, but finding a neutral ground on which many languages and analyses could function. We decided to use a term-level language for linear, polarized, intuitionistic logic taking the form of an abstract machine. The linear aspect of this machine enabled a native treatment of resources within a type system, as opposed to adding resource-awareness ad-hoc to already established ones. The polarized aspect allowed programs to carry over the operational semantics of their source languages into the machine through the paradigm of Call-by-Push-Value. Finally, the abstract-machine-as-a-term-language paradigm at the core of our method put us in an excellent position to embed logical reasoning within programs, as mainstream logic could be used at type-level to statically reasons about the invariants of programs.

This choice was made in light of the fact that studying code is the only possible source of bounds over all possible runs of a program, and that the Curry-Howard correspondence describes the representations of programs most able to incorporate the logical reasoning

backing the corpus of algorithm analysis. This did center our work onto a particular paradigm for source languages, namely typed functional programming. But his was not unfortunate, as providing all-encompassing compatibility with most existing programming paradigms would be a babelian endeavor. Armed with this basis, our research shed light on some aspects of the problem that we would now like to recount.

10.1 Insights

First and foremost, the use of the polarized abstract machine enabled a formalization of resource footprint as a runtime invariant, namely the total amount of resource available to the program. A well-typed program is then one which is endowed with enough resources. In an analogy to physics (in a closed system, energy is preserved), the amount of resources held by a program is always constant, and therefore an invariant. Furthermore, the use of a small-step semantics, in which each command determines an atomic evaluation step, reduces the problem to one of resource preservation at each command. We have shown that as programs run, the approximate, statically-determined models to the constraints they generate becomes more and more precise, as more and more *possible* assignments of type-level parameters are replaced by assignments determined by what occurred *in actuality* at runtime.

In the same vein, the use of linear types to encode resources enabled us not only to guarantee the total amount doesn't change at runtime, but also to centralize them in a unique token. The preservation and non-duplication of resources is then dependent only on that one token, and it is easier to prove that resources are preserved at runtime than with a non-linear system in which resources may furthermore be spread around. This allowed for a simpler proof of resource preservation.

Lastly, the use of an explicit continuation stack and shifts allows programs to embed their own evaluation contexts and control flow. This means a syntax-directed approach suffices to understand when control-flow boundaries are passed and to yield resources accordingly. Explicit stacks can be manipulated the same way data structures are, offering a uniform treatment of potential and information flow through data and computations alike. Those three ingredients are the essence of the resource-passing-as-an-parameterized-effect transformation that enriches program with resource manipulations.

Moving on from the machine, the Call-by-Push-Value formalism was found to be well-suited for resource analysis, as the explicit thunks and closures act as natural control-and-resource flow boundaries within programs. This allows for an automated translation of programming languages whose operational semantics are based on term rewriting. Furthermore, the placement of thunks and closures suffices to further elaborate resource-aware semantics. This eases the formulation of resource analyses for those languages, which is a further benefit as it allows many languages to use the same implementation of the analyzer.

On the other side of the analysis, we have split the type-and-resources analysis common to AARA into the collection of local information and the exploitation of that data through constraint solving. To do so, we relied on first-order logic as a logical neutral ground. This allowed to export resource constraint to SMT solvers and proof assistants. The ability to separate solvers and typecheckers allows reasoning on constraints to be reusable across type systems. We have provided such constraint post-processing to check programs for a given bound, and to solve the specific polynomial optimizations problems used by RAML. The ability to switch solvers depending on the kind of analysis being undertaken opens avenues for new complexity classes to be analysed, while keeping the infrastructure creating the constraint unchanged. This would enable more analyses to be preformed without changing frontends.

10.2 Further work

As we mentioned in the previous section, having a reusable, modular framework for resource analysis of typed functional languages opens up new avenues of work, extending the range of its application and increasing its precision.

We first intend to finish the development of our prototype analyzer *AutoBill*, and extend it with feature capable of making it a fully automated resource analyzer. Implementing a more powerful type system in a standard manner will, we believe, lead a more straightforward a resilient implementation than one starting from a Hindley-Milner type system and then adding parameters.

There are, of course, some theoretical avenues as well. First of which, is the upgrade of the intermediate representation used from the abstract machine, a term-language for intuitionistic linear logic, what we called the to one for full linear logic. This corresponds, on the operational

level, to the addition of multiple continuations to the term language. The full power of multiple continuations does allow for the implementation of the `call/cc` function from scheme, but reductions of that full power are also of interest. For example, the use of affine continuations (i.e. which cannot be shared), allows for so-called "multi-barreled functions", which can return to many possible continuations, and way be used to implement exceptions. Furthermore, if those affine continuations can be used in data structures and stack constructors, effect handler systems such as the one implemented in OCaml can be implemented and analysed. This could open the way for resource analysis in the presence of effects handlers, and may open the possibility of analysis in the presence of user-defined effects. Currently, supporting user-defined effects would require analyzing monadic primitives, which suffers from the high-order argument issue mentioned in chapter 8.

Lastly, we would like to mention the possibility of extending the constraint solving system used in AutoBill. A first direction for that extension would be to allow for the establishment of poly-logarithmic bounds on complexity. This would have an obvious and welcome impact to our results, as it would allow for a wide class of efficient data structures and algorithms to be tested. As we mentioned in this manuscript, an AARA indexing scheme was presented by Hoffmann for this purpose, but remains unimplemented in the literature. Another, related direction, would be the addition of reasoning capabilities on trees. This is currently a thorny point for analysis, as such reasoning does not fit within the finite-size patterns framework employed by AARA. As such, bounding the height of a tree, for example, is not possible when said tree grown above a hard-coded height. This is of course related to the topic of poly-logarithmic complexity, and the inference of finer invariants at type level. A tree maintaining an optimal height through self-balancing would maintain a logarithmic height relative to its size, and seeing this at compile-time will require new form of invariant encoding out of reach of current methods.

10.3 Final words

As our work was undergoing, we often reflected on the fact that static memory bounds for programs relied on a wide range of information, which in turn relied on a diverse corpus of analyses. In an efficient data structure, sizes and shapes are heavily dependent on subtle invariants, which are often tailored to the structure itself. Likewise, sophisticated control

flow is also common to make code more efficient, which creates other difficulties. Obtaining a resource bound therefore requires reconstructing invariants along data and computations, without which no memory bound can be created. In that sense, memory bounding is a "cherry on the cake" that can only be obtained after comprehensive knowledge of a program is achieved. As simple as the problem is to state is its solution complex.

It was never an option to obtain bounds on arbitrary programs, but we have strained to demand, as have others before us, as little efforts of our potential users as would be feasible. The maximal empowerment of programmers remains the cardinal virtue of those creating development tools and static analyses. Paradoxically, it is that very freedom which, structurally, puts results out of reach. While we do affirm that any and all shortcoming in our work is our responsibility alone, as we close this manuscript, we would like to take a step back on this virtue and the "social contract" it creates between us and programmers.

The freedom of programmer to write the program they wish to write is in obvious tension with any ability to provide guarantees that would satisfy them. On the one hand, more work is required to obtain tooling that will, one day, provide those bounds. On the other hand, asking little of programmers, who are the most knowledgeable about their program, is to the analysis designer, a strange form of self-denial. This is especially true as the situations where those bounds matter most are exactly those we might hope to encounter principled approaches to programming.

When operational safety is required – when those bounds are more desirable – it could very well be the case that the freedom of programmers to create is not constrained, but enabled by the ability to obtain finer guarantees through more explicit information. The same processes that cause some programmer to choose typed approaches when no such obligations is put on them, for both cultural and pragmatic reasons, might also enable a wider success of resource analysis in the future.

Appendix A

Syntax, Typing Rules, Reductions

A.1 Mini-ML

A.1.1 Syntax

⟨*program*⟩ ::= ⟨*toplevel*⟩ ...

⟨*toplevel*⟩ ::= ⟨*def*⟩ | ⟨*expr*⟩

⟨*def*⟩ ::= `type (` [*A* `,`...] `)` *K* `=` ⟨*consdef*⟩ [...]
- | `let` *x* `=` ⟨*expr*⟩
- | `let rec` *f x* ... `=` ⟨*expr*⟩ [`and` ...]

⟨*consdef*⟩ ::= `|` *k* [`of` ⟨*type*⟩ `*` ...]

⟨*type*⟩ ::= *A*
- | ⟨*type*⟩ → ⟨*type*⟩
- | *K* [`(` ⟨*type*⟩ `,` ... `)`]

⟨*val*⟩ ::= ⟨*intlitt*⟩
- | *x*
- | *k* [`(` ⟨*val*⟩ `,` ... `)`]
- | `fun` *x* `->` ⟨*expr*⟩
- | `rec` *x* `=` ⟨*val*⟩ [`and` ...] `in` ⟨*val*⟩

⟨*intlitt*⟩ ::= `0` | `1` | `2` | ...

⟨*intop*⟩ ::= `add` | `sub` | `eq` | `le` | ...

⟨*expr*⟩ ::= `x`
- | ⟨*intlitt*⟩
- | ⟨*intop*⟩
- | ⟨*expr*⟩ `:` ⟨*type*⟩
- | `do` ⟨*stmt*⟩
- | *k* [`(` ⟨*expr*⟩ `,` ... `)`]
- | `match` ⟨*val*⟩ `with` ⟨*clause*⟩ [...] `end`
- | `fun` *x* `->` ⟨*expr'*⟩
- | ⟨*expr*⟩ ⟨*expr*⟩
- | `let` *x* `=` ⟨*expr*⟩ `in` ⟨*expr*⟩
- | `let rec` *x* `=` ⟨*val*⟩ [`and` ...] `in` ⟨*expr*⟩
- | `tick` ⟨*intlitt*⟩ `in` ⟨*expr*⟩

⟨*clause*⟩ ::= `|` *k* [`(` *x* `,` ... `)`] `->` ⟨*expr*⟩

⟨*expr'*⟩ ::= ⟨*expr*⟩ (without *stmt* sub-terms)

⟨*stmt*⟩ ::= `return` ⟨*expr'*⟩
- | `let` *x* `<-` ⟨*stmt*⟩ `;` ⟨*stmt*⟩
- | `let` *x* `=` ⟨*expr'*⟩ `;` ⟨*stmt*⟩
- | `let mut` *x* `:=` ⟨*expr'*⟩ `;` ⟨*stmt*⟩
- | *x* `:=` ⟨*expr'*⟩ `;` ⟨*stmt*⟩
- | `if` ⟨*expr'*⟩ `then` ⟨*stmt*⟩ [`else` ⟨*stmt*⟩] `end` `;` ⟨*stmt*⟩
- | `for` *x* `in` ⟨*expr'*⟩ `do` ⟨*stmt*⟩ `done` `;` ⟨*stmt*⟩
- | `return!` ⟨*expr'*⟩
- | `continue!`
- | `break!`

A.2 CBPV-ML

A.2.1 Syntax

⟨*program*⟩ ::= ⟨*toplevel*⟩ . . . ⟨*expr*⟩

⟨*toplevel*⟩ ::=
- | `data` K`[(`A`:±,...)] = [`⟨*consdef*⟩ . . . `]`
- | `comput` K`[(`A`:± , ...)] = [`⟨*methdef*⟩ . . . `]`
- | `type` K `[(`A `: ± , ...)] =` ⟨*type*⟩
- | `let` x `=` ⟨*expr*⟩

⟨*consdef*⟩ ::= | k`[(`⟨*type*⟩`,...)]`

⟨*methdef*⟩ ::= | k`[(`⟨*type*⟩`,...)] ->` ⟨*type*⟩

⟨*type*⟩ ::= A | K`[(`⟨*type*⟩`,...)]` | `!` ⟨*type*⟩

⟨*intlitt*⟩ ::= `0` | `1` | `2` | . . .

⟨*intop*⟩ ::= `add` | `sub` | `eq` | `le` | . . .

⟨*clause*⟩ ::= | k `([`x`,...]) ->` ⟨*expr*⟩

⟨*val*⟩ ::= ⟨*intlitt*⟩
- | x
- | k`([`⟨*val*⟩`,...])`
- | `closure` ⟨*expr*⟩
- | `rec` x `=` ⟨*expr*⟩
- | `exp` ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*intop*⟩
- | ⟨*expr*⟩ `:` ⟨*type*⟩
- | `get` ⟨*clause*⟩ . . . `end`
- | ⟨*expr*⟩`.`k`([`⟨*val*⟩`,...])`
- | `match` ⟨*val*⟩ `with` ⟨*clause*⟩ . . . `end`
- | `let` x `=` ⟨*val*⟩ `in` ⟨*expr*⟩
- | `tick` ⟨*intlitt*⟩ `in` ⟨*expr*⟩
- | `thunk` ⟨*val*⟩
- | `force` x `=` ⟨*expr*⟩ `in` ⟨*expr*⟩
- | `open` ⟨*val*⟩
- | `unfold` ⟨*val*⟩
- | `unexp` ⟨*val*⟩

A.2.2 Reduction

$$\texttt{let}\, x = V\, \texttt{in}\, e \longrightarrow_0 e[V/x]$$

$$\texttt{open}\, \texttt{closure}(e) \longrightarrow_0 e$$

$$\texttt{unfold}\, \texttt{rec}\, y = e \longrightarrow_0 e[(\texttt{rec}\, y = e)/y]$$

$$\texttt{force}\, x = e\, \texttt{in}\, e'' \longrightarrow_k \texttt{force}\, x = e'\, \texttt{in}\, e'' \qquad \textit{when } e \rightarrow_k e'$$

$$\texttt{force}\, x = \texttt{thunk}\, V\, \texttt{in}\, e \longrightarrow_0 e[V/x]$$

$$\texttt{unexp}\, \texttt{exp}\, e \longrightarrow_0 e$$

$$\texttt{tick}\, k\, \texttt{in}\, e \longrightarrow_k e$$

$$\begin{array}{l}\texttt{match}\, k\, (\, V_1\, ,\, \ldots\, ,\, V_n\,)\, \texttt{with} \\ \quad \mid k\, (\, x_1\, ,\, \ldots\, ,\, x_n\,)\, \texttt{->}\, e \\ \quad \mid \ldots \\ \texttt{end}\end{array} \longrightarrow_0 e[V_1/x_1, \ldots, V_n/x_n]$$

$$e.k(V_1,\ \ldots,\ V_n) \longrightarrow_k e'.k(V_1,\ \ldots,\ V_n) \qquad \textit{when } e \rightarrow_k e'$$

$$\textit{primop}\, .\texttt{call}(n,m,\ldots) \longrightarrow_0 \texttt{thunk}\, p \qquad \textit{(integer primitives)}$$

$$\begin{array}{l}(\,\texttt{get} \\ \quad \mid k\, (\, x_1\, ,\, \ldots\, ,\, x_n\,)\, \texttt{->}\, e \\ \quad \mid \ldots \\ \texttt{end}\,)\, .\, k\, (\, V_1\, ,\, \ldots\, ,\, V_n\,)\end{array} \longrightarrow_0 e[V_1/x_1, \ldots, V_n/x_n]$$

A.2.3 Embedding of Mini-ML

$$
\begin{aligned}
[\![A]\!]_{type} &= !A \\
\left[\!\left[K(\vec{T})\right]\!\right]_{type} &= !K(\overrightarrow{[\![T]\!]_{type}}) \\
[\![T_1 \to T_2]\!]_{type} &= !\Downarrow([\![A]\!]_{type} \multimap \Uparrow[\![B]\!]_{type})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fun\text{-}decl} &= \texttt{comput}\,(A : +) \to (B : -) = \texttt{call}\,(!A) \to B \\
\left[\!\left[\overrightarrow{decl}; e\right]\!\right]_{prog} &= \mathit{fun\text{-}decl};\ \overrightarrow{[\![decl]\!]_{toplevel}};\ [\![e]\!] \\
\left[\!\left[\,\texttt{type}\,K(\vec{A}) = \overrightarrow{k(\vec{T})}\right]\!\right]_{toplevel} &= \texttt{data}\ K(\overrightarrow{A : +}) = \overrightarrow{k(\overrightarrow{[\![T]\!]_{type}})} \\
[\![\,\texttt{let}\,x = e]\!]_{toplevel} &= \texttt{let}\,x = (\,\texttt{let}\,y = \texttt{force}\,[\![e]\!]\,\texttt{in}\,\texttt{exp}\,y)
\end{aligned}
$$

$$
\begin{aligned}
[\![V]\!] &= \texttt{thunk}\,\texttt{exp}\,[\![V]\!]_{val} \\
[\![\,\texttt{let}\,x = e\,\texttt{in}\,e']\!] &= \texttt{let}\,x = \texttt{force}\,[\![e]\!]\,\texttt{in}\,[\![e']\!] \\
[\![\,\texttt{tick}\,k\,\texttt{in}\,e]\!] &= \texttt{tick}\,k\,\texttt{in}\,[\![e]\!] \\
[\![k(\vec{e_i})]\!] &= \overrightarrow{\texttt{force}\,x_i = [\![e_i]\!]\,\texttt{in}}\ \texttt{thunk}\,\texttt{exp}\,[\![k(\vec{x_i})]\!]_{val} \\
\left[\!\left[\,\texttt{match}\,e\,\texttt{with}\,\overrightarrow{k(\vec{x}) \to e'}\right]\!\right] &= \texttt{force}\,y = [\![e]\!]\,\texttt{in} \\
&\quad\ \texttt{force}\,z = \texttt{unexp}\,[\![y]\!]\,\texttt{in} \\
&\quad\ \texttt{match}\,z\,\texttt{with}\,\overrightarrow{k(\vec{x}) \to [\![e']\!]} \\
[\![e_1\,e_2]\!] &= \texttt{force}\,x_1 = [\![e_1]\!]\,\texttt{in} \\
&\quad\ \texttt{force}\,x_2 = [\![e_2]\!]\,\texttt{in} \\
&\quad\ (\,\texttt{unexp}\,x_1).\,\texttt{call}\,(x_2)
\end{aligned}
$$

$$
[\![\,\texttt{let rec}\,\overrightarrow{x_i = e_i}]\!]_{toplevel} = \begin{cases} \texttt{let}\,y = \left(\,\texttt{rec}\,y = \texttt{get}\,\left(\overrightarrow{k_i() \to [\![e_i]\!]\left[\overrightarrow{get_i(y)\,/\,x_i}\right]}\right)\right) \\ \overrightarrow{\texttt{let}\,x_i = get_i(y)} \end{cases}
$$

$$
[\![\,\texttt{let rec}\,\overrightarrow{x_i = e_i}\,\texttt{in}\,e']\!] = \begin{cases} \texttt{let}\,y = \left(\,\texttt{rec}\,y = \texttt{get}\,\left(\overrightarrow{k_i() \to [\![e_i]\!]\left[\overrightarrow{get_i(y)\,/\,x_i}\right]}\right)\right)\ \texttt{in} \\ \overrightarrow{\texttt{let}\,x_i = get_i(y)}\,\texttt{in} \\ [\![e']\!] \end{cases}
$$

$$
\begin{aligned}
get_i(y) &= \texttt{exp}\,((\,\texttt{unfold}\,y).k_i()) \\
\mathit{rec\text{-}decl}(\vec{x_i}) &= \texttt{comput}\,K(\overrightarrow{A_i} : -) = \overrightarrow{k_i() \to A_i}
\end{aligned}
$$

A.3 System-L

A.3.1 Expression-level Syntax

Sort-level

$$\mathbf{b} ::= \mathbf{pos} \mid \mathbf{neg} \qquad \textit{(base type sorts)}$$
$$\mathbf{m} ::= \mathbf{b} \mid \mathbf{s} \to \mathbf{m} \qquad \textit{(monotypes sorts)}$$
$$\mathbf{k} ::= (\mathbf{m}_1, \cdots, \mathbf{m}_n) \to \mathbf{m} \qquad \textit{(type constructors sorts)}$$

Parameter-level

$\mathbf{s}$ *(parameter sort variables)*
R *(relation variables)*
φ *(operator variables)*
$\tau ::= \alpha \mid \varphi(\vec{\tau})$ *(parameters)*
$E ::= \top \mid \tau = \tau' \wedge E \mid R(\vec{\tau}) \wedge E$ *(equations)*
$C ::= \top \mid \bot \mid E \wedge C \mid E \Rightarrow C \mid \forall(\alpha : \mathbf{s}).C \mid \exists(\alpha : \mathbf{s}).C$

Type-level

T *(monotype variables)*
K *(type constructor variables including **fix**, !, ⇓, ⇑)*
$A ::= T \mid\mid K(\vec{A})(\vec{\tau}) \mid \lambda(\alpha : \mathbf{s}).A \mid A(\tau)$ *(monotypes)*

Expression-level

x *(value variables)*
a *(stack variables)*
k *(constructor variables)*
$\sigma ::= \varepsilon \mid [/x]\varepsilon \mid [x/y]\varepsilon$ *(substitutions)*
$\Gamma ::= \varepsilon \mid x : A, \Gamma$ *(value scopes)*
$\Delta ::= a : A$ *(stack scopes)*
$c ::= \langle t^+ \parallel S^+ \rangle \mid \langle V^- \parallel e^- \rangle \mid \langle \sigma ; c \rangle \mid \langle \$n ; c \rangle$ *(commands)*
$t^+ ::= \mu a^+.c \mid V^+$ *(terms)*
$e^- ::= \mu x^-.c \mid S^-$ *(environments)*
$V^+ ::= x^+ \mid k_{\vec{\tau}}(\overrightarrow{V^+}) \mid \mathbb{0}_\Gamma \mid \Downarrow(V^-) \mid \mu!a^-.c$ *(pos. values)*
$S^+ ::= a^+ \mid \mu x^-.c \mid \mu k_{\vec{\alpha}}(\overrightarrow{\overrightarrow{x^+}}).c \mid \mu\mathbb{0}_{\Gamma,\Delta} \mid \mu\Downarrow a^-.c \mid ! \cdot S^-$ *(pos. stacks)*
$V^- ::= x^- \mid \mu a^-.c \mid \mu k_{\vec{\alpha}}(\overrightarrow{\overrightarrow{x^+}; a^-}).c \mid \mu\top \mid \mu\Uparrow(a^-).c$ *(neg. values)*
$\mid \mu\ \mathbf{fix}(a^-).\langle \mathbf{self} \parallel S^- \rangle$ *(neg. values, cont.)*
$S^- ::= a^- \mid k_{\vec{\tau}}(\overrightarrow{V^+}) \cdot S^- \mid \top_{\Gamma,\Delta} \mid \Uparrow \cdot S^+ \mid \mathbf{fix} \cdot S^-$ *(neg. stacks)*

A.3.2 Type definitions syntax

type $K(A_1 : \mathbf{m_1}, \ldots, A_n : \mathbf{m_n})$ $(\alpha_1 : \mathbf{s_1}, \ldots, \alpha_m : \mathbf{s_m})$: $\mathbf{b} = B$

newtype $K(A_1 : \mathbf{m_1}, \ldots, A_n : \mathbf{m_n})$ $(\alpha_1 : \mathbf{s_1}, \ldots, \alpha_m : \mathbf{s_m})$: $\mathbf{b} = k$ **of** B
where $\beta_1 : \mathbf{s'_1}, \ldots, \beta_p : \mathbf{s'_p}$ **with** E

data $K(\vec{\mathsf{A}} : \vec{\mathbf{m}})$ $(\vec{\alpha} : \vec{\mathbf{s}})$ =
| k_1 **of** $\mathsf{B}_{1,1} \otimes \cdots \otimes \mathsf{B}_{1,k_1}$ **where** $\overrightarrow{\beta_1} : \overrightarrow{\mathbf{s}'_1}$ **with** E_1
...
| k_m **of** $\mathsf{B}_{m,1} \otimes \cdots \otimes \mathsf{B}_{m,k_m}$ **where** $\overrightarrow{\beta_m} : \overrightarrow{\mathbf{s}'_m}$ **with** E_m
end

comput $K(\vec{\mathsf{A}} : \vec{\mathbf{m}})$ $(\vec{\alpha} : \vec{\mathbf{s}})$ =
| k_1 **of** $\mathsf{B}_{1,1} \otimes \cdots \otimes \mathsf{B}_{1,k_1} \multimap D_1$ **where** $\overrightarrow{\beta_1} : \overrightarrow{\mathbf{s}'_1}$ **with** E_1
...
| k_m **of** $\mathsf{B}_{m,1} \otimes \cdots \otimes \mathsf{B}_{m,k_m} \multimap D_m$ **where** $\overrightarrow{\beta_m} : \overrightarrow{\mathbf{s}'_m}$ **with** E_m
end

- Parameters sorts, relations, and operators are implicitly in scope in all definitions.
- Type constructor definitions are mutually recursive: all type constructors are implicitly in scope with their declared sort in all type-level definition, including their own.
- A **type** definition is valid whenever $\overrightarrow{A : \mathbf{m}}, \overrightarrow{\alpha : \mathbf{s}} \vdash B : \mathbf{b}$.
- A **newtype** definition is valid whenever $\overrightarrow{A : \mathbf{m}}, \overrightarrow{\alpha : \mathbf{s}}, \overrightarrow{\beta : \mathbf{s'}} \vdash B : \mathbf{b}$ and $\overrightarrow{\alpha : \mathbf{s}}, \overrightarrow{\beta : \mathbf{s'}} \vdash E$.
- A **data** or **comput** definition in valid whenever $\Theta_i \vdash B_{i,j} : \mathbf{pos}$ and $\Theta_i \vdash D_i : \mathbf{neg}$ for all $i \leq m$ and $j \leq k_i$, and $\Theta_i = \overrightarrow{A : \mathbf{m}}, \overrightarrow{\alpha : \mathbf{s}}, \overrightarrow{\beta_i : \mathbf{s_i}}$.
- For each type constructor K,the judgment $\vec{k} \twoheadrightarrow K$ is valid whenever the family of constructors $\vec{k}$ contains all constructors used in the definition of K, without duplicates.
- For each constructor k^+ (resp. k^-) of a data (resp. computation) type K, the judgment

$$k : \exists \vec{\beta}.\, E \wedge \vec{B} \to K(\vec{A})(\vec{\alpha})$$

$$k : \exists \vec{\beta}.\, E \wedge \vec{B} \multimap D \to K(\vec{A})(\vec{\alpha})$$

is valid for the variables in the definitions of k in K and every instantiation thereof of the variables $\vec{A}$ as monotypes and $\vec{\alpha}$ as parameters.

A.3.3 Type-level Sorting

Generalities All parameters sort, relation, and operator definitions are implicitly in scope in all sorting rules. The scope used is $\Theta = \overrightarrow{A : \mathbf{m}}, \overrightarrow{\alpha : \mathbf{s}}$.

Parameters ($\Theta \vdash \tau : \mathbf{s}$ and $\Theta \vdash E$)

$$\frac{}{\Theta, \alpha : \mathbf{s} \vdash \alpha : \mathbf{s}}\ \textbf{(param-var)} \qquad \frac{\overrightarrow{\Theta \vdash \tau : \mathbf{s}} \qquad \varphi : \vec{\mathbf{s}} \to \mathbf{s}'}{\Theta \vdash \varphi(\vec{\tau}) : \mathbf{s}'}\ \textbf{(param-op)}$$

$$\frac{\overrightarrow{\Theta \vdash \tau : \mathbf{s}} \qquad \Theta \vdash E \qquad R : \vec{\mathbf{s}}}{\Theta \vdash R(\vec{\tau}) \wedge E}\ \textbf{(param-rel)} \qquad \frac{\Theta \vdash \tau_i : \mathbf{s} \qquad \Theta \vdash E}{\Theta \vdash \tau_1 = \tau_2 \wedge E}\ \textbf{(param-eq)}$$

$$\frac{}{\Theta \vdash \top}\ \textbf{(param-top)}$$

Constraints ($\Theta \vdash C$)

$$\frac{}{\Theta \vdash \top}\ \textbf{(constr-top)} \qquad \frac{}{\Theta \vdash \bot}\ \textbf{(constr-top)}$$

$$\frac{\Theta, \alpha : \mathbf{s} \vdash C}{\Theta \vdash \forall(\alpha : \mathbf{s}).C}\ \textbf{(constr-forall)} \qquad \frac{\Theta, \alpha : \mathbf{s} \vdash C}{\Theta \vdash \exists(\alpha : \mathbf{s}).C}\ \textbf{(constr-exists)}$$

$$\frac{\Theta \vdash E \qquad \Theta \vdash C}{\Theta \vdash E \wedge C}\ \textbf{(constr-and)} \qquad \frac{\Theta \vdash E \qquad \Theta \vdash C}{\Theta \vdash E \Rightarrow C}\ \textbf{(constr-arrow)}$$

Monotypes ($\Theta \vdash A : \mathbf{m}$)

$$\frac{}{\Theta, T : \mathbf{m} \vdash T : \mathbf{m}}\ \textbf{(type-var)}$$

$$\frac{\overrightarrow{\Theta \vdash A : \mathbf{m}} \qquad \overrightarrow{\Theta \vdash \tau : \mathbf{s}} \qquad K : \vec{\mathbf{m}} \times \vec{\mathbf{s}} \to \mathbf{b}}{\Theta \vdash K(\vec{A})(\vec{\tau}) : \mathbf{b}}\ \textbf{(type-cons)}$$

$$\frac{\Theta, \overrightarrow{\alpha : \mathbf{s}} \vdash A : \mathbf{b}}{\Theta \vdash \lambda(\overrightarrow{\alpha : \mathbf{s}}).A : \vec{\mathbf{s}} \to \mathbf{b}}\ \textbf{(type-abs)} \qquad \frac{\Theta \vdash A : \vec{\mathbf{s}} \to \mathbf{b} \qquad \overrightarrow{\Theta \vdash \tau : \mathbf{s}}}{\Theta \vdash A(\vec{\tau}) : \mathbf{b}}\ \textbf{(type-app)}$$

A.3.4 Simple Type System

Generalities

- Parameter-level terms are not allowed in the simple type system. Parameters, simple constraints and constraints are to be considered erased from type constructor definitions. The parameter-level $\vec{\tau}$ and $\vec{\alpha}$ in constructor applications and pattern matching are likewise erase. The type-level terms $\lambda\alpha.A$ and $A(\tau)$ are forbidden.
- The sorting of type variables in the simple type system is determined by polarity annotations: every type annotated $(-)^+$ (resp. $(-)^-$) is of sort **pos** (resp. **neg**).
- In the command $\langle\sigma;\, c\rangle$ and the **(struct)** rule, the application $\Gamma\sigma$ of the variable-for-variable substitution σ to a value scope Γ is defined by the following equations, and undefined otherwise:

$$\begin{aligned}\Gamma\varepsilon &= \Gamma\\ (\Gamma, x :!A)([/x]\sigma) &= \Gamma\sigma\\ (\Gamma, x :!A)([\vec{y}/x]\sigma) &= \overrightarrow{y :!A}, (\Gamma\sigma) \quad \textit{if } x \textit{ doesnt appear in } \sigma.\end{aligned}$$

Judgments Judgments use a value-variable scope Γ associates to value-variables x a type of sort **pos** or **neg**. It is linear, of arbitrary size, and always taken up to permutation of bindings. There also is a stack-variable scope Δ that associates to a single stack variable a a type of sort **pos** of **neg**. Linearity doesn't apply to this one-variable context.

Commands	$c :$	$(\Gamma$	$\vdash \Delta \quad)$
Terms		Γ	$\vdash t \ : A$
Values		Γ	$\vdash V : A$
Right clauses		Γ	$\vdash k(\vec{x}).c : A$ **cl.**
Environments		$\Gamma \mid e$	$: A \vdash \Delta$
Stacks		$\Gamma \mid S$	$: A \vdash \Delta$
Left clauses		$\Gamma \mid k(\vec{x}; a).c$	$: A \vdash \Delta$

Commands and binders

$$\frac{\Gamma \vdash t : A^{\pm} \qquad \Gamma' \mid e : A^{\pm} \vdash \Delta}{\langle t \parallel e \rangle^{\pm} : (\Gamma, \Gamma' \vdash \Delta)}\ \textbf{(cut)}$$

$$\frac{c : (\Gamma \vdash \Delta) \qquad n \in \mathbb{Z}}{\langle \$n;\, c \rangle : (\Gamma \vdash \Delta)}\ \textbf{(cost)} \qquad \frac{c : (\Gamma\sigma \vdash \Delta)}{\langle \sigma;\, c \rangle : (\Gamma \vdash \Delta)}\ \textbf{(struct)}$$

$$\frac{c : (\Gamma, x : A^{\pm} \vdash \Delta)}{\Gamma \mid \mu x.c : A^{\pm} \vdash \Delta}\ \textbf{(bind-V)} \qquad \frac{c : (\Gamma \vdash a : A^{\pm})}{\Gamma \vdash \mu a.c : A^{\pm}}\ \textbf{(bind-S)}$$

Values

$$\frac{}{\Gamma, x : A^{\pm} \vdash x : A^{\pm}}\ \textbf{(id-R)}$$

$$\frac{\overrightarrow{\Gamma \vdash V : B^{+}} \qquad k : \vec{B} \to K(\vec{A})^{+}}{\vec{\Gamma} \vdash k(\vec{V}) : K(\vec{A})^{+}}\ \textbf{(data-R)}$$

$$\frac{\overrightarrow{\Gamma \vdash k(\vec{x}; a).c : K(\vec{A})^{-}\ \textbf{cl}.} \qquad \vec{k} \twoheadrightarrow K}{\Gamma \vdash \mu(\overrightarrow{k(\vec{x}; a).c}) : K(\vec{A})^{-}}\ \textbf{(comput-R)}$$

$$\frac{c : (\Gamma, \overrightarrow{x : B^{+}} \vdash \Delta) \qquad k : \vec{B} \to K(A^{+})^{+}}{\Gamma \mid k(\vec{x}).c : K(\vec{A^{+}})^{+} \vdash \Delta\ \textbf{cl}.}\ \textbf{(clause-L)}$$

No rule for $\mathbb{0}$

$$\frac{}{\Gamma \vdash \mu \top_{\Gamma} : \top^{-}}\ \textbf{(top-R)}$$

$$\frac{\Gamma \vdash V : A^{-}}{\Gamma \vdash \Downarrow V : \Downarrow (A^{-})^{+}}\ \textbf{(closure-R)}$$

$$\frac{c : (\Gamma \vdash a : A^{+})}{\Gamma \vdash \mu \Uparrow a.c : \Uparrow (A^{+})^{-}}\ \textbf{(thunk-R)}$$

$$\frac{c : (!\Gamma \vdash a : A^{-})}{\Gamma \mu !a.c : !(A^{-})^{+}}\ \textbf{(exp-R)}$$

$$\frac{c : (!\Gamma \mid S : \mathbf{fix}\, A^{-} \vdash a : A^{-})}{\Gamma \vdash \mu \mathbf{fix}(a).\langle \mathbf{self} \parallel S \rangle : (\mathbf{fix}\, A^{-})^{-}}\ \textbf{(fix-R)}$$

Stacks

$$\frac{}{\Gamma \mid a : A^{\pm} \vdash a : A^{\pm}}\ \textbf{(id-L)}$$

$$\frac{\overrightarrow{\Gamma \mid k(\vec{x}).c : K(\vec{A})^{+} \vdash \Delta\ \mathbf{cl}.} \qquad \vec{k} \twoheadrightarrow K}{\Gamma \mid \mu(\overrightarrow{k(\vec{x}).c}) : K(\vec{A})^{+} \vdash \Delta}\ \textbf{(data-L)}$$

$$\frac{c : (\Gamma, \overrightarrow{x : B^{+}} \vdash a : D^{-}) \qquad k : \overrightarrow{B^{+}} \multimap D^{-} \to K(A^{+})^{-}}{\Gamma \vdash k(\vec{x}; a).c : K(A^{+})^{-}\ \mathbf{cl}.}\ \textbf{(clause-R)}$$

$$\frac{\overrightarrow{\Gamma \vdash V : A^{+}} \qquad \Gamma' \mid S : D^{-} \vdash \Delta \qquad k : (\overrightarrow{A^{+}}; D^{-}) \to K(B^{+})^{-}}{\vec{\Gamma}, \Gamma' \mid k(\vec{V}) \cdot S : K(B^{+})^{-} \vdash \Delta}\ \textbf{(comput-L)}$$

$$\frac{}{\Gamma \mid \mu 0_{\Gamma,\Delta} \vdash \Delta}\ \textbf{(zero-L)}$$

No rule for $\top$

$$\frac{c : (\Gamma, x : A^{-} \vdash \Delta)}{\Gamma \mid \mu{\Downarrow}x.c : {\Downarrow}(A^{-})^{+} \vdash \Delta}\ \textbf{(closure-L)}$$

$$\frac{\Gamma \mid S : A^{+} \vdash \Delta}{\Gamma \mid \Uparrow \cdot S : \Uparrow(A^{+})^{-} \vdash \Delta}\ \textbf{(thunk-L)}$$

$$\frac{\Gamma \mid S : A^{-} \vdash \Delta}{\Gamma \mid ! \cdot S : (!A^{-})^{+} \vdash \Delta}\ \textbf{(exp-L)}$$

$$\frac{\Gamma \mid S : A^{-} \vdash \Delta}{\Gamma \mid \mathbf{fix} \cdot S : (\mathbf{fix}\, A^{-})^{-} \vdash \Delta}\ \textbf{(fix-L)}$$

A.3.5 Parameterized Type System

Generalities Sorting annotations are omitted from the rules for legibility. The polarity of types remain unchanged from the simple type system. Readers may refer to its rules to see polarity annotations. As for parameter sorts, they are fixed by the sorting judgment of constructors k and the sorting rule of the equality constraint $\tau = \tau'$.

Judgements Judgments use a parameter scope Θ assigning to parameter variables α a parameter-level sort $\mathbf{s}$, a constraint C such that $\Theta \vdash C$, and value and stack scopes Γ and Δ as above, whose assigned types A satisfy $\Theta \vdash A : \mathbf{b}$.

Commands	$c : (\Theta \models C) \triangleright (\Gamma \vdash \Delta)$
Terms	$(\Theta \models C) \triangleright \Gamma \vdash t : A$
Values	$(\Theta \models C) \triangleright \Gamma \vdash V : A$
Right clauses	$(\Theta \models C) \triangleright \Gamma \vdash k(\vec{x}).c : A$ **cl.**
Environments	$(\Theta \models C) \triangleright \Gamma \mid e : A \vdash \Delta$
Stacks	$(\Theta \models C) \triangleright \Gamma \mid S : A \vdash \Delta$
Left clauses	$(\Theta \models C) \triangleright \Gamma \mid k(\vec{x}; a).c : A \vdash \Delta$ **cl.**

Parameter-level rules The rules are expressed once for a generic sequent $(\Theta \models C) \triangleright (\Gamma \vdash \Delta)$ as opposed many times, once for each specific typing judgment above.

$$\frac{(\Theta \models C) \triangleright (\Gamma \vdash \Delta) \qquad \mathrm{fv}(\Gamma, \Delta, C) \subset \Theta}{(\Theta, \Theta' \models C) \triangleright (\Gamma \vdash \Delta)}\ \textbf{(fol-weak)}$$

$$\frac{(\Theta \models C') \triangleright (\Gamma \vdash \Delta) \qquad \Theta \models C \Rightarrow C'}{(\Theta \models C) \triangleright (\Gamma \vdash \Delta)}\ \textbf{(fol-sub)}$$

$$\frac{(\Theta \models C \wedge \alpha = \alpha) \triangleright (\Gamma \vdash \Delta)}{(\Theta \models C) \triangleright (\Gamma \vdash \Delta)}\ \textbf{(fol-refl)}$$

$$\frac{(\Theta \models C\theta) \triangleright (\Gamma\theta \vdash \Delta\theta) \qquad \theta = \mathrm{mgu}(\tau = \tau')}{(\Theta, x, y \models C \wedge \tau = \tau') \triangleright (\Gamma \vdash \Delta)}\ \textbf{(fol-unify)}$$

Rules from the simple type system Most rules in the parameterized type system are transparent at sequent-level. For those rules, a generic translation scheme is applied to the corresponding simply-typed rule. This scheme adds a fixed parameter-level fragment $(\Theta \models C)$

to each sequent in a rule. This expressed below for a generic rule between generic sequents $(\Theta \models C) \triangleright (\Gamma \vdash \Delta)$ as opposed specific typing judgments. Those generic sequents are then instantiated for each syntactic sort of expression in a given simply-typed rule.

$$\frac{(\Gamma_1 \vdash \Delta_1) \;\; \ldots \;\; (\Gamma_n \vdash \Delta_n) \;\; \textit{(side rules)}}{(\Gamma' \vdash \Delta')}$$

$$\downarrow$$

$$\frac{(\Theta \models C) \triangleright (\Gamma_1 \vdash \Delta_1) \qquad \ldots \qquad (\Theta \models C) \triangleright (\Gamma_n \vdash \Delta_n) \qquad \textit{(side rules)}}{(\Theta \models C) \triangleright (\Gamma' \vdash \Delta')}$$

This is used to defined the parameterized versions of the left and right rules **(id)**, **(closure)**, **(thunk)**, **(exp)**, **(fix)**, and **(bind)**, and well as of the **(cut)**, **(cost)**, and **(struct)** rules.

Values (new rules)

$$\frac{\overrightarrow{(\Theta \models C) \triangleright \Gamma \vdash V : B} \qquad \vdash k : \exists \vec{\alpha}.E \wedge \vec{B} \to K(\vec{A})(\vec{\tau})}{(\vec{\Theta} \models \exists \vec{\alpha}.\, E \wedge \vec{C} \wedge \overrightarrow{\beta = \tau'}) \triangleright \vec{\Gamma} \vdash k^{\vec{}}_{\vec{\tau'}}(\vec{V}) : K(\vec{A})(\vec{\tau'})} \textbf{(data-R)}$$

$$\frac{\vdash k : \exists \vec{\beta}.E \wedge \vec{B} \multimap D \to K(\vec{A})(\vec{\tau}) \qquad c : (\Theta, \vec{\alpha} \models C) \triangleright (\Gamma, \overrightarrow{x : B} \vdash a : D)}{(\Theta \models \forall \vec{\alpha}.\, E \Rightarrow C) \triangleright \Gamma \vdash k_{\vec{\alpha}}(\vec{x}; a).c : K(\vec{A})(\vec{\tau})\ \textbf{cl}.} \textbf{(clause-R)}$$

$$\frac{\vdash \vec{k} \twoheadrightarrow K \qquad \overrightarrow{(\Theta \models C) \triangleright \Gamma \mid k_\alpha(\vec{x}; a).c : K(\vec{A})(\vec{\tau})\ \textbf{cl}.}}{(\vec{\Theta} \models \vec{C}) \triangleright \Gamma \vdash \mu(\overrightarrow{k_{\vec{\alpha}}(\vec{x}; a).c}) : K(\vec{A})(\vec{\tau})} \textbf{(comput-R)}$$

$$\frac{}{(\Theta \models \top) \triangleright \Gamma \vdash \mu\top_\Gamma : \top}\ \textbf{(top-R)} \qquad \frac{}{(\Theta \models \bot) \triangleright \Gamma \vdash \mathbb{0}_\Gamma : \mathbb{0}}\ \textbf{(zero-R)}$$

Stacks (new rules)

$$\frac{\vdash k : \exists \vec{\beta}.E \wedge \vec{B} \to K(\vec{A})(\vec{\tau}) \qquad c : (\Theta, \vec{\alpha} \models C) \triangleright (\Gamma, \overrightarrow{x : B} \vdash \Delta)}{(\Theta \models \forall \vec{\alpha}.\, E \Rightarrow C) \triangleright \Gamma \mid k_{\vec{\alpha}}(\vec{x}).c : K(\vec{A})(\vec{\tau}) \vdash \Delta \ \mathbf{cl}.} \ \textbf{(clause-L)}$$

$$\frac{\vdash \vec{k} \twoheadrightarrow K \qquad \overrightarrow{(\Theta \models C) \triangleright \Gamma \mid k_\alpha(\vec{x}).c : K(\vec{A})(\vec{\tau}) \vdash \Delta \ \mathbf{cl}.}}{(\vec{\Theta} \models \vec{C}) \triangleright \Gamma \mid \mu(\overrightarrow{k_{\vec{\alpha}}(\vec{x}).c}) : K(\vec{A})(\vec{\tau}) \vdash \Delta} \ \textbf{(data-L)}$$

$$\frac{\begin{array}{c}\overrightarrow{(\Theta \models C) \triangleright \Gamma \vdash V : B} \\ (\Theta' \models C') \triangleright \Gamma' \mid S : D \vdash \Delta \qquad k : \exists \vec{\alpha}.E \wedge \vec{B} \multimap D \to K(A)(\tau)\end{array}}{(\vec{\Theta}, \Theta' \models \vec{C} \wedge C' \wedge \overrightarrow{\alpha = \tau}) \triangleright \vec{\Gamma}, \Gamma' \mid k_{\vec{\tau}}(\vec{V}) \cdot S : K(A)(\vec{\tau}) \vdash \Delta} \ \textbf{(comput-L)}$$

$$\frac{}{(\Theta \models \top) \triangleright \Gamma \mid \mu\mathbb{0}_{\Gamma,\Delta} : \mathbb{0} \vdash \Delta} \ \textbf{(zero-L)} \qquad \frac{}{(\Theta \models \bot) \triangleright \Gamma \mid \top_{\Gamma,\Delta} : \top \vdash \Delta} \ \textbf{(top-L)}$$

A.3.6 Reduction

$$\langle \mu a^{\pm}.c \parallel S^{\pm} \rangle^{\pm} \triangleright c[S/a] \qquad \textbf{(bind-}S\textbf{)}$$

$$\langle V^{\pm} \parallel \mu x^{\pm}.c \rangle^{\pm} \triangleright c[V/x] \qquad \textbf{(bind-}V\textbf{)}$$

$$\langle \$k; c \rangle \triangleright_k c \qquad \textbf{(cost)}$$

$$\langle \sigma; c \rangle \triangleright c\sigma \qquad \textbf{(struct)}$$

$$\langle k^i_{\vec{\tau}}(\vec{V}) \parallel \mu(\overrightarrow{k^j_{\vec{\beta_j}}(\vec{x_j}).c_j}) \rangle^{+} \triangleright c_i[\overrightarrow{V_i/x_i}, \overrightarrow{\tau/\beta_i}] \qquad \textbf{(data)}$$

$$\langle \mu(\overrightarrow{k^i_{\vec{\beta_i}}(\vec{x_i}; a_i).c_i}) \parallel k^j_{\vec{\tau}}(\vec{V}) \cdot S^{-} \rangle^{-} \triangleright c_j[\overrightarrow{V/x_j}, S/a_j, \overrightarrow{\tau/\beta_j}] \qquad \textbf{(comput)}$$

$$\langle \mathbb{0}_{\Gamma} \parallel \mu \mathbb{0}_{\Gamma',\Delta} \rangle^{+} \triangleright c, \text{ for any } c : (\Gamma, \Gamma' \vdash \Delta) \qquad \textbf{(zero)}$$

$$\langle \mu \top_{\Gamma} \parallel \top_{\Gamma',\Delta} \rangle^{-} \triangleright c \qquad \textbf{(top)}$$

$$\langle \Downarrow V \parallel \mu \Downarrow x.c \rangle \triangleright c[V/x] \qquad \textbf{(closure)}$$

$$\langle \mu \Uparrow (a).c \parallel \Uparrow \cdot S \rangle \triangleright c[S/a] \qquad \textbf{(thunk)}$$

$$\langle \mu !a.c \parallel ! \cdot S \rangle^{+} \triangleright x[S/a] \qquad \textbf{(exp)}$$

$$\langle \mu \textbf{fix}(a).\langle \textbf{self} \parallel S \rangle \parallel \textbf{fix} \cdot S^{-} \rangle^{-} \triangleright \langle \mu \textbf{fix}(a).\langle \textbf{self} \parallel S \rangle \parallel S[S'/a] \rangle^{-} \qquad \textbf{(fix)}$$

$$\mu x^{\pm}.\langle x^{\pm} \parallel e^{\pm} \rangle^{\pm} \triangleright^{e} e \qquad \textbf{(eta-bind-}V\textbf{)}$$

$$\mu a^{\pm}.\langle t^{\pm} \parallel a^{\pm} \rangle^{\pm} \triangleright^{t} t \qquad \textbf{(eta-bind-}S\textbf{)}$$

$$\mu(\overrightarrow{k^j_{\vec{\beta_j}}(\vec{x_j}).\langle k^j_{\vec{\beta_j}}(\vec{x_j}) \parallel e^{+} \rangle}) \triangleright^{t} e^{+} \qquad \textbf{(eta-data)}$$

$$\mu(\overrightarrow{k^i_{\vec{\tau_i}}(\vec{x_i}; a_i).\langle t^{-} \parallel k^i_{\vec{\tau_i}}(\vec{x_i}) \cdot a_i \rangle}) \triangleright^{t} t^{-} \qquad \textbf{(eta-comput)}$$

$$\mu \mathbb{0}_{\Gamma,\Delta} \triangleright^{e} e^{+}, \text{ for any } \Gamma \mid e : \mathbb{0} \vdash \Delta \qquad \textbf{(eta-zero)}$$

$$\mu \top_{\Gamma} \triangleright^{t} t^{-}, \text{ for any } \Gamma \vdash t : \top \qquad \textbf{(eta-top)}$$

$$\mu \Downarrow x.\langle \Downarrow x \parallel e^{+} \rangle \triangleright^{e} e^{+} \qquad \textbf{(eta-closure)}$$

$$\mu \Uparrow a.\langle t \parallel \Uparrow \cdot a \rangle \triangleright^{t} t^{-} \qquad \textbf{(eta-thunk)}$$

$$\mu !a.\langle t^{+} \parallel ! \cdot a \rangle^{+} \triangleright^{t} t \qquad \textbf{(eta-exp)}$$

$$\mu \textbf{fix}(a).\langle \textbf{self} \parallel \mu x.\langle [/x]; \langle t^{-} \parallel \textbf{fix} \cdot a \rangle \rangle \triangleright^{t} t^{-}$$

A.3.7 Embedding of CBPV-ML

CBPV-ML Values $V \longmapsto$ System-L values $[\![V]\!]$

$$
\begin{aligned}
[\![x]\!]_{val} &= x \\
[\![n]\!]_{val} &= n \\
\left[\!\left[k(\vec{V})\right]\!\right]_{val} &= k(\overrightarrow{[\![V]\!]}) \\
[\![\,\texttt{closure}\,e]\!]_{val} &= \Downarrow [\![e]\!] \\
[\![\,\texttt{rec}\,x = e]\!]_{val} &= \mu(\mathbf{self} \cdot a).\langle \mathbf{self} \parallel \mu x.\langle [\![e]\!] \parallel a\rangle\rangle \\
[\![\,\texttt{exp}\,V]\!]_{val} &= \mu !a.\langle [\![V]\!] \parallel a\rangle
\end{aligned}
$$

CBPV-ML expressions $e \longmapsto$ System-L terms* $[\![e]\!]$

$$
\begin{aligned}
[\![\,\texttt{let}\,x = V\,\texttt{in}\,e]\!] &= \mu a.\langle [\![V]\!] \parallel \mu x.\langle [\![e]\!] \parallel a\rangle\rangle \\
[\![\,\texttt{tick}\,k\,\texttt{in}\,e]\!] &= \mu a.\langle \$k; \langle [\![e]\!] \parallel a\rangle\rangle \\
[\![\,\texttt{thunk}\,V]\!] &= \mu \Uparrow a.\langle [\![V]\!] \parallel a\rangle \\
[\![\,\texttt{force}\,x = e\,\texttt{in}\,e']\!] &= \mu a.\langle [\![e]\!] \parallel \Uparrow \cdot \mu x.\langle [\![e']\!] \parallel a\rangle\rangle \\
[\![\,\texttt{open}\,V]\!] &= \mu a.\langle [\![V]\!] \parallel \mu \Downarrow x.\langle x \parallel a\rangle\rangle \\
[\![\,\texttt{unfold}\,V]\!] &= \mu a.\langle [\![V]\!] \parallel \mathbf{fix} \cdot a\rangle \\
[\![\,\texttt{unexp}\,V]\!] &= \mu a.\langle [\![V]\!] \parallel\, ! \cdot a\rangle \\
[\![op]\!]_{val} &= op \\
[\![V]\!] &= [\![V]\!]_{val} \\
\left[\!\left[\,\texttt{get}\,\overrightarrow{k(\vec{x}) \to e}\,\texttt{end}\,\right]\!\right] &= \mu(\overrightarrow{k(\vec{x}; a).\langle [\![e]\!] \parallel a\rangle}) \\
\left[\!\left[e.k(\vec{V})\right]\!\right] &= \mu a.\langle [\![e]\!] \parallel k(\overrightarrow{[\![V]\!]}) \cdot a\rangle \\
\left[\!\left[\,\texttt{match}\,V\,\texttt{with}\,\overrightarrow{k(\vec{x}) \to e}\right]\!\right] &= \mu a.\langle [\![V]\!] \parallel \mu(\overrightarrow{k(\vec{x}).\langle [\![e]\!] \parallel a\rangle})\rangle
\end{aligned}
$$

**Compiling CBPV-ML expression also involves adding an explicit sharing $\langle \sigma\ ;\ \cdots \rangle$ to each newly created term, which is elided here for simplicity.*

Bibliography

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. "Cost Analysis of Java Bytecode". In: *Programming Languages and Systems*. Ed. by R. De Nicola. Berlin, Heidelberg: Springer, 2007, pp. 157–172. ISBN: 978-3-540-71316-6. DOI: 10.1007/978-3-540-71316-6_12 (cit. on p. 182).

[2] E. Albert, D. E. Alonso-Blas, P. Arenas, J. Correas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, A. N. Masud, G. Puebla, J. M. Rojas, et al. "Automatic Inference of Bounds on Resource Consumption". In: *Formal Methods for Components and Objects: 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures*. Ed. by E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue. Berlin, Heidelberg: Springer, 2013, pp. 119–144. ISBN: 978-3-642-40615-7. DOI: 10.1007/978-3-642-40615-7_4 (cit. on p. 35).

[3] E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, and G. Román-Díez. "Object-Sensitive Cost Analysis for Concurrent Objects". In: *Software Testing, Verification and Reliability* 25.3 (2015), pp. 218–271. ISSN: 1099-1689. DOI: 10.1002/stvr.1569 (cit. on p. 35).

[4] E. Albert, P. Arenas, S. Genaim, and G. Puebla. "A Practical Comparator of Cost Functions and Its Applications". In: *Science of Computer Programming*. Special Issue on Foundational and Practical Aspects of Resource Analysis (FOPARA) 2009 & 2011 111 (Nov. 1, 2015), pp. 483–504. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.12.001 (cit. on p. 36).

[5] E. Albert, P. Arenas, S. Genaim, and G. Puebla. "Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis". In: *Static Analysis*. Ed. by M. Alpuente and G. Vidal. Berlin, Heidelberg: Springer, 2008, pp. 221–237. ISBN: 978-3-540-69166-2. DOI: 10.1007/978-3-540-69166-2_15 (cit. on p. 197).

[6] E. Albert, P. Arenas, S. Genaim, and G. Puebla. "Cost Relation Systems: A Language-Independent Target Language for Cost Analysis". In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the Eighth Spanish Conference on Programming and Computer Languages (PROLE 2008) 248 (Aug. 5, 2009), pp. 31–46. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2009.07.057 (cit. on p. 182).

[7] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. "Cost Analysis of Object-Oriented Bytecode Programs". In: *Theoretical Computer Science*. Quantitative Aspects of Programming Languages (QAPL 2010) 413.1 (Jan. 6, 2012), pp. 142–159. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2011.07.009 (cit. on p. 182).

[8] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. "COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode". In: *Formal Methods for Components and Objects*. Ed. by F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever. Berlin, Heidelberg: Springer, 2008, pp. 113–132. ISBN: 978-3-540-92188-2. DOI: 10.1007/978-3-540-92188-2_5 (cit. on p. 182).

[9] E. Albert, M. Bofill, C. Borralleras, E. Martin-Martin, and A. Rubio. "Resource Analysis Driven by (Conditional) Termination Proofs". In: *Theory and Practice of Logic Programming* 19.5-6 (Sept. 2019), pp. 722–739. ISSN: 1471-0684, 1475-3081. DOI: 10.1017/S1471068419000152 (cit. on p. 36).

[10] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio. "*Don't Run on Fumes*— Parametric Gas Bounds for Smart Contracts". In: *Journal of Systems and Software* 176 (June 1, 2021), p. 110923. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.110923 (cit. on p. 35).

[11] E. Albert, S. Genaim, and M. Gómez-Zamalloa. "Heap Space Analysis for Garbage Collected Languages". In: *Science of Computer Programming* 78.9 (Sept. 1, 2013), pp. 1427–1448. ISSN: 0167-6423. DOI: 10.1016/j.scico.2012.10.008 (cit. on p. 36).

[12] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. "Syntax-Guided Synthesis". In: *2013 Formal Methods in Computer-Aided Design*. 2013 Formal Methods in Computer-Aided Design. Oct. 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385 (cit. on p. 200).

[13] J.-M. ANDREOLI. "Logic Programming with Focusing Proofs in Linear Logic". In: *Journal of Logic and Computation* 2.3 (June 1, 1992), pp. 297–347. ISSN: 0955-792X. DOI: 10.1093/logcom/2.3.297 (cit. on p. 48).

[14] S. Bellantoni and S. Cook. "A New Recursion-Theoretic Characterization of the Polytime Functions (Extended Abstract)". In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing - STOC '92*. The Twenty-Fourth Annual ACM Symposium. Victoria, British Columbia, Canada: ACM Press, 1992, pp. 283–293. ISBN: 978-0-89791-511-3. DOI: 10.1145/129712.129740 (cit. on p. 186).

[15] D. Binder, M. Tzschentke, M. Müller, and K. Ostermann. *Grokking the Sequent Calculus (Functional Pearl)*. June 20, 2024. arXiv: 2406.14719 [cs]. URL: http://arxiv.org/abs/2406.14719 (visited on 06/24/2024). Pre-published (cit. on pp. 58, 184).

[16] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*

- *POPL '77*. The 4th ACM SIGACT-SIGPLAN Symposium. Los Angeles, California: ACM Press, 1977, pp. 238–252. DOI: 10.1145/512950.512973 (cit. on p. 34).

[17] P.-L. Curien, M. Fiore, and G. Munch-Maccagnoni. "A Theory of Effects and Resources: Adjunction Models and Polarised Calculi". In: *Proc. POPL*. 2016. DOI: 10.1145/2837614.2837652 (cit. on pp. 58, 69, 76).

[18] P.-L. Curien and H. Herbelin. "The Duality of Computation". In: *SIGPLAN Not.* 35.9 (Sept. 1, 2000), pp. 233–243. ISSN: 0362-1340. DOI: 10.1145/357766.351262 (cit. on p. 58).

[19] P.-L. Curien and G. Munch-Maccagnoni. "The Duality of Computation under Focus". In: *Theoretical Computer Science*. Ed. by C. S. Calude and V. Sassone. Vol. 323. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 165–181. ISBN: 978-3-642-15239-9 978-3-642-15240-5. DOI: 10.1007/978-3-642-15240-5_13 (cit. on p. 58).

[20] U. Dal Lago and M. Gaboardi. "Linear Dependent Types and Relative Completeness". In: *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. 2011 IEEE 26th Annual Symposium on Logic in Computer Science. June 2011. DOI: 10.1109/LICS.2011.22 (cit. on pp. 27, 187).

[21] U. Dal lago and B. Petit. "The Geometry of Types". In: *ACM SIGPLAN Notices (POPL)* (Jan. 23, 2013). ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2480359.2429090 (cit. on pp. 27, 28, 30, 187).

[22] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and J. Rehof. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 170).

[23] P. Downen, L. Maurer, Z. M. Ariola, and S. Peyton Jones. "Sequent Calculus as a Compiler Intermediate Language". In: *ACM SIGPLAN Notices* 51.9 (Dec. 5, 2016), pp. 74–88. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/3022670.2951931 (cit. on p. 58).

[24] T. Ehrhard. "Call-By-Push-Value from a Linear Logic Point of View". In: *Programming Languages and Systems*. Ed. by P. Thiemann. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2016, pp. 202–228. ISBN: 978-3-662-49498-1. DOI: 10.1007/978-3-662-49498-1_9 (cit. on p. 90).

[25] G. Gentzen. "Untersuchungen über das logische Schließen. I". In: *Mathematische Zeitschrift* 39.1 (Dec. 1, 1935), pp. 176–210. ISSN: 1432-1823. DOI: 10.1007/BF01201353 (cit. on p. 109).

[26] G. Gentzen. "Untersuchungen über das logische Schließen. II". In: *Mathematische Zeitschrift* 39.1 (Dec. 1, 1935), pp. 405–431. ISSN: 1432-1823. DOI: 10.1007/BF01201363 (cit. on p. 109).

[27] J.-Y. Girard. *Le point aveugle: cours de logique*. Paris: Hermann, 2007. ISBN: 978-2-7056-6633-0 978-2-7056-6634-7 (cit. on pp. 9, 48).

[28] J.-Y. Girard. "Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types". In: *Studies in Logic and the Foundations of Mathematics*. Ed. by J. E. Fenstad. Vol. 63. Proceedings of the Second Scandinavian Logic Symposium. Elsevier, Jan. 1, 1971, pp. 63–92. DOI: 10.1016/S0049-237X(08)70843-7 (cit. on p. 165).

[29] J. Grosen, D. M. Kahn, and J. Hoffmann. *Automatic Amortized Resource Analysis with Regular Recursive Types*. Apr. 26, 2023. DOI: 10.48550/arXiv.2304.13627. arXiv: 2304.13627 [cs]. Pre-published (cit. on p. 194).

[30] J. Herbrand. *Recherches Sur La Théorie de La Démonstration*. Vol. 33. J. Dziewulski, 1930 (cit. on p. 38).

[31] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. *An Overview of Ciao and Its Design Philosophy*. Feb. 27, 2011. DOI: 10.48550/arXiv.1102.5497. arXiv: 1102.5497 [cs]. Pre-published (cit. on p. 35).

[32] R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 0002-9947. DOI: 10.2307/1995158. JSTOR: 1995158 (cit. on p. 165).

[33] J. Hoffmann, K. Aehlig, and M. Hofmann. "Multivariate Amortized Resource Analysis". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '11*. The 38th Annual ACM SIGPLAN-SIGACT Symposium. Austin, Texas, USA: ACM Press, 2011, p. 357. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926427 (cit. on p. 20).

[34] J. Hoffmann and S. Jost. "Two Decades of Automatic Amortized Resource Analysis". In: *Mathematical Structures in Computer Science* (Mar. 16, 2022), pp. 1–31. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129521000487 (cit. on pp. 19, 20, 185, 186, 191).

[35] M. Hofmann. "A Mixed Modal/Linear Lambda Calculus with Applications to Bellantoni-Cook Safe Recursion". In: *Computer Science Logic*. Ed. by M. Nielsen and W. Thomas. Red. by G. Goos, J. Hartmanis, and J. Van Leeuwen. Vol. 1414. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 275–294. ISBN: 978-3-540-64570-2 978-3-540-69353-6. DOI: 10.1007/BFb0028020 (cit. on p. 185).

[36] M. Hofmann. "A Type System for Bounded Space and Functional In-Place Update". In: *Nordic Journal of Computing* 7.4 (Dec. 1, 2000), pp. 258–289. ISSN: 1236-6064 (cit. on pp. 8, 186).

[37] M. Hofmann. "Linear Types and Non-Size-Increasing Polynomial Time Computation". In: *Information and Computation*. International Workshop on Implicit Computational Complexity (ICC'99) 183.1 (May 25, 2003), pp. 57–85. ISSN: 0890-5401. DOI: 10.1016/S0890-5401(03)00009-9 (cit. on pp. 19, 185).

[38] M. Hofmann. "Programming Languages Capturing Complexity Classes". In: *ACM SIGACT News* 31.1 (Mar. 2000), pp. 31–42. ISSN: 0163-5700. DOI: 10.1145/346048.346051 (cit. on p. 186).

[39] M. Hofmann. "Semantics of Linear/Modal Lambda Calculus". In: *Journal of Functional Programming* 9.3 (May 1999), pp. 247–277. ISSN: 09567968. DOI: 10.1017/S0956796899003433 (cit. on p. 185).

[40] M. Hofmann and S. Jost. "Static Prediction of Heap Space Usage for First-Order Functional Programs". In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '03. New York, NY, USA: Association for Computing Machinery, Jan. 15, 2003, pp. 185–197. ISBN: 978-1-58113-628-9. DOI: 10.1145/604131.604148 (cit. on p. 186).

[41] M. Hofmann, L. Leutgeb, D. Obwaller, G. Moser, and F. Zuleger. "Type-Based Analysis of Logarithmic Amortised Complexity". In: *Mathematical Structures in Computer Science* 32.6 (June 2022), pp. 794–826. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129521000232 (cit. on p. 193).

[42] S. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields. "Practical Type Inference for Arbitrary-Rank Types". In: *Journal of Functional Programming* 17.1 (Jan. 2007), pp. 1–82. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796806006034 (cit. on p. 165).

[43] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. "Static Determination of Quantitative Resource Usage for Higher-Order Programs". In: *SIGPLAN Not.* 45.1 (Jan. 17, 2010), pp. 223–236. ISSN: 0362-1340. DOI: 10.1145/1707801.1706327 (cit. on p. 186).

[44] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. ""Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis". In: *FM 2009: Formal Methods*. Ed. by A. Cavalcanti and D. R. Dams. Berlin, Heidelberg: Springer, 2009, pp. 354–369. ISBN: 978-3-642-05089-3. DOI: 10.1007/978-3-642-05089-3_23 (cit. on p. 186).

[45] S. Jost, P. Vasconcelos, M. Florido, and K. Hammond. "Type-Based Cost Analysis for Lazy Functional Languages". In: *Journal of Automated Reasoning* 59.1 (June 1, 2017), pp. 87–120. ISSN: 1573-0670. DOI: 10.1007/s10817-016-9398-9 (cit. on p. 186).

[46] D. M. Kahn and J. Hoffmann. "Exponential Automatic Amortized Resource Analysis". In: *Foundations of Software Science and Computation Structures: 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, Apr. 25, 2020, pp. 359–380. ISBN: 978-3-030-45230-8. DOI: 10.1007/978-3-030-45231-5_19 (cit. on p. 192).

[47] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall Software Series. Englewood Cliffs, N.J: Prentice-Hall, 1978. 228 pp. ISBN: 978-0-13-110163-0 (cit. on p. 5).

[48] T. Knoth, D. Wang, A. Reynolds, J. Hoffmann, and N. Polikarpova. "Liquid Resource Types". In: *Proceedings of the ACM on Programming Languages* 4 (ICFP Aug. 2, 2020), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3408988 (cit. on p. 198).

[49] J.-L. Krivine. "A Call-by-Name Lambda-Calculus Machine". In: *Higher-Order and Symbolic Computation* 20.3 (Nov. 20, 2007), pp. 199–207. ISSN: 1388-3690, 1573-0557. DOI: 10.1007/s10990-007-9018-9 (cit. on p. 28).

[50] U. D. Lago. "Implicit Computation Complexity in Higher-Order Programming Languages: A Survey in Memory of Martin Hofmann". In: *Mathematical Structures in Computer Science* 32.6 (June 2022), pp. 760–776. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129521000505 (cit. on p. 186).

[51] P. J. Landin. "The Mechanical Evaluation of Expressions". In: *The Computer Journal* 6.4 (Jan. 1, 1964), pp. 308–320. ISSN: 0010-4620. DOI: 10.1093/comjnl/6.4.308 (cit. on p. 28).

[52] P. B. Levy. "Adjunction Models For Call-By-Push-Value With Stacks". In: *Electronic Notes in Theoretical Computer Science* 69 (Feb. 2003), pp. 248–271. ISSN: 15710661. DOI: 10.1016/S1571-0661(04)80568-1 (cit. on pp. 61, 76, 89).

[53] P. B. Levy. *Call-By-Push-Value*. Dordrecht: Springer Netherlands, 2003. ISBN: 978-94-010-3752-5 978-94-007-0954-6. DOI: 10.1007/978-94-007-0954-6 (cit. on p. 89).

[54] P. B. Levy. "Call-by-Push-Value: A Subsuming Paradigm". In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 1999, pp. 228–243 (cit. on p. 89).

[55] S. Liang, P. Hudak, and M. Jones. "Monad Transformers and Modular Interpreters". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '95*. The 22nd ACM SIGPLAN-SIGACT Symposium. San Francisco, California, United States: ACM Press, 1995, pp. 333–343. ISBN: 978-0-89791-692-9. DOI: 10.1145/199448.199528 (cit. on p. 98).

[56] B. Lichtman and J. Hoffmann. "Arrays and References in Resource Aware ML". In: *International Conference on Formal Structures for Computation and Deduction (FSCD)*. Ed. by D. Miller. 2017. ISBN: 978-3-95977-047-7. DOI: 10.4230/LIPIcs.FSCD.2017.26 (cit. on p. 186).

[57] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo. "Interval-Based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption". In: *Theory and Practice of Logic Programming* 18.2 (Mar. 2018), pp. 167–223. ISSN: 1471-0684, 1475-3081. DOI: 10.1017/S1471068418000042 (cit. on pp. 35, 197).

[58] D. Marker. *Model Theory*. Vol. 217. Graduate Texts in Mathematics. New York: Springer-Verlag, 2002. ISBN: 978-0-387-98760-6. DOI: 10.1007/b98860 (cit. on p. 110).

[59] M. Méndez-Lojo, J. Navas, and M. V. Hermenegildo. "A Flexible, (C)LP-Based Approach to the Analysis of Object-Oriented Programs". In: *Logic-Based Program Synthesis and Transformation*. Ed. by A. King. Berlin, Heidelberg: Springer, 2008, pp. 154–168. ISBN: 978-3-540-78769-3. DOI: 10.1007/978-3-540-78769-3_11 (cit. on p. 35).

[60] R. Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Sciences* 17.3 (Dec. 1, 1978), pp. 348–375. ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90014-4 (cit. on p. 4).

[61] R. Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Sciences* 17.3 (Dec. 1, 1978), pp. 348–375. ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90014-4 (cit. on p. 165).

[62] E. Moggi. "Computational Lambda-Calculus and Monads". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS)*. Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS). 1989. ISBN: 978-0-8186-1954-0. DOI: 10.1109/LICS.1989.39155 (cit. on p. 98).

[63] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. "User-Definable Resource Usage Bounds Analysis for Java Bytecode". In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the Fourth Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2009) 253.5 (Dec. 1, 2009), pp. 65–82. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2009.11.015 (cit. on p. 35).

[64] T. Nipkow. "Amortized Complexity Verified". In: *Interactive Theorem Proving*. Ed. by C. Urban and X. Zhang. Cham: Springer International Publishing, 2015, pp. 310–324. ISBN: 978-3-319-22102-1. DOI: 10.1007/978-3-319-22102-1_21 (cit. on p. 194).

[65] Y. Niu and J. Hoffmann. "Automatic Space Bound Analysis for Functional Programs with Garbage Collection". In: LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, pp. 543–521. DOI: 10.29007/xkwx (cit. on p. 186).

[66] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. "Simple Unification-Based Type Inference for GADTs". In: *ACM SIGPLAN Notices* 41.9 (Sept. 16, 2006), pp. 50–61. ISSN: 0362-1340. DOI: 10.1145/1160074.1159811 (cit. on p. 156).

[67] F. Pottier and Didier Rémy. "The Essence of ML Type Inference". In: *Advanced Topics in Types and Programming Languages*. Ed. by Pierce, Benjamin C. The MIT Press, Jan. 2005, pp. 389–489. ISBN: 978-0-262-16228-9 (cit. on pp. 159, 167).

[68] F. Pottier and Y. Régis-Gianas. "Stratified Type Inference for Generalized Algebraic Data Types". In: *SIGPLAN Not.* 41.1 (Jan. 11, 2006), pp. 232–244. ISSN: 0362-1340. DOI: 10.1145/1111320.1111058 (cit. on p. 156).

[69] V. Rajani, M. Gaboardi, D. Garg, and J. Hoffmann. "A Unifying Type-Theory for Higher-Order (Amortized) Cost Analysis". In: *Proceedings of the ACM on Programming*

Languages 5 (POPL Jan. 4, 2021), 27:1–27:28. DOI: 10.1145/3434308 (cit. on pp. 32, 33, 187).

[70] J. C. Reynolds. "Towards a Theory of Type Structure". In: *Programming Symposium*. Ed. by B. Robinet. Berlin, Heidelberg: Springer, 1974, pp. 408–425. ISBN: 978-3-540-37819-8. DOI: 10.1007/3-540-06859-7_148 (cit. on p. 165).

[71] P. M. Rondon, M. Kawaguci, and R. Jhala. "Liquid Types". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08: ACM SIGPLAN Conference on Programming Language Design and Implementation. Tucson AZ USA: ACM, June 7, 2008, pp. 159–169. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375602 (cit. on p. 198).

[72] A. Serrano, P. Lopez-Garcia, and M. V. Hermenegildo. "Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types". In: *Theory and Practice of Logic Programming* 14.4-5 (July 2014), pp. 739–754. ISSN: 1471-0684, 1475-3081. DOI: 10.1017/S147106841400057X (cit. on pp. 35, 197).

[73] R. J. Simmons. *Structural Focalization*. Mar. 16, 2014. DOI: 10.48550/arXiv.1109.6273. arXiv: 1109.6273 [cs]. Pre-published (cit. on p. 48).

[74] D. D. Sleator and R. E. Tarjan. "Self-Adjusting Binary Search Trees". In: *J. ACM* 32.3 (July 1, 1985), pp. 652–686. ISSN: 0004-5411. DOI: 10.1145/3828.3835 (cit. on p. 194).

[75] R. E. Tarjan. "Amortized Computational Complexity". In: *SIAM Journal on Algebraic Discrete Methods* (Apr. 1985). ISSN: 0196-5212, 2168-345X. DOI: 10.1137/0606031 (cit. on p. 14).

[76] A. Tarski. "The Concept of Truth in Formalized Languages". In: *Logic, Semantics, Metamathematics*. Ed. by A. Tarski. Clarendon Press, 1956, pp. 152–278 (cit. on p. 108).

[77] S. Ullrich and de Moura. *Supplement of "'do' Unchained: Embracing Local Imperativity in a Purely Functional Language"*. Zenodo, June 1, 2022. DOI: 10.5281/ZENODO.6684085 (cit. on pp. 100, 182).

[78] S. Ullrich and L. de Moura. "'Do' Unchained: Embracing Local Imperativity in a Purely Functional Language (Functional Pearl)". In: *Proceedings of the ACM on Programming Languages* 6 (ICFP Aug. 29, 2022), pp. 512–539. ISSN: 2475-1421. DOI: 10.1145/3547640 (cit. on pp. 100, 182).

[79] N. Vazou, P. M. Rondon, and R. Jhala. "Abstract Refinement Types". In: *Programming Languages and Systems*. Ed. by M. Felleisen and P. Gardner. Berlin, Heidelberg: Springer, 2013, pp. 209–228. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_13 (cit. on p. 198).

[80] J. B. Wells. "Typability and Type Checking in System F Are Equivalent and Undecidable". In: *Annals of Pure and Applied Logic* 98.1 (June 30, 1999), pp. 111–156. ISSN: 0168-0072. DOI: 10.1016/S0168-0072(98)00047-5 (cit. on p. 165).

[81] M. V. Wilkes. "Computers Then and Now". In: *Journal of the ACM* 15.1 (Jan. 1, 1968), pp. 1–7. ISSN: 0004-5411. DOI: 10.1145/321439.321440 (cit. on p. 3).