A Reusable Machine-Calculus for Automated Resource Analyses

Hector Suzanne, Emmanuel Chailloux

A reusable machine-calculus for automated resource analyses*

Hector Suzanne[0000−0002−7761−8779] and Emmanuel Chailloux[0000−0002−2400−9523]

Sorbonne Université – CNRS, LIP6
{hector.suzanne,emmanuel.chailloux}@lip6.fr

Abstract. We introduce an automated resource analysis technique is introduced, targeting a Call-By-Push-Value abstract machine, with memory prediction as a practical goal. The machine has a polymorphic and linear type system enhanced with a first-order logical fragment, which encodes both low-level operational semantics of resource manipulations and high-level synthesis of algorithmic complexity.
Resource analysis must involve a diversity of static analysis, for escape, aliasing, algorithmic invariants, and more. Knowing this, we implement the *Automated Amortized Resource Analysis* framework (AARA) from scratch in our generic system. In this setting, access to resources is a state-passing effect which produces a compile-time approximation of run-time resource usage.
We implemented type inference constraint generation for our calculus, accompanied with an elaboration of bounds for iterators on algebraic datatypes, for minimal ML-style programming languages with Call-by-Value and Call-By-Push-Value semantics. The closed-formed bounds are derived as multivariate polynomials over the integers. This now serves as a base for the development of an experimental toolkit for automated memory analysis of functional languages.

Keywords: Type Theory · Static Analysis · Memory Consumption · Amortized analysis · Call-by-Push-Value.

1 Introduction

Typed functional programming offers some structural safety out-of-the-box, but correctness of systems also depends on quantitative, material concerns: memory consumption must remain within bounds, latency and energy cost must be low, etc. This highlights the need for general-purpose resource analysis tools for typed functional languages. But functional languages in the style of ML or Haskell pose specific challenges for resource predictions. First, dynamic allocations is an inherent problem, since the size of the allocated data cannot be fully determined statically. Second, prevalent use of garbage collectors and reference counting in

* This work has been partially performed at the IRILL center for Free Software Research and Innovation in Paris, France.

those languages means de-allocation points are implicit. Lastly, closures (and high-order programming in general) purposefully hide the amount of resources and state of data they close over.

Amortized complexity [16] has been used to extend functional type systems with resource analyses, notably in Hoffmann's *Automated Amortized Resource Analysis (AARA)* [4]. But extending both the precision and generality of current methods puts a large burden on formalisms and implementations. In this paper, we develop a three-step approach to improve this situation:

1. We extend a Call-by-Push-Value abstract machine typed with intuitionistic linear logic introduced by Curien et al. Programs are decomposed as interaction between values and stacks, whose types can involve *parameters* variables, which denotes quantities of resources and number of combinatorial patterns in data. These parameters are guarded by first-order constraints. Call-by-Push-Value semantics strictly partition types into data-types and computation-types, and use *closures* and *thunks* to mediate between them.
2. Then, we devise a Call-By-Push-Value effects, which reflects at type-level the logical requirements inputs and outputs states of programs: execution go well for all states ($\forall$) with enough resources, and returns some arbitrary state ($\exists$) with some allocated resources.
3. Finally, our implementation extracts a global constraint on resource usage from the type of the rewritten program. Those final constraints are expressed in first-order arithmetic, and can be exported or solved automatically using our heuristic mimicking AARA reasoning. Solvers restricted to intuitionistic logic can elaborate resource expression back into the program.

Users can annotate higher-order code with domain-specific constraints which partially specify runtime behavior. This allows for verification of high-order programs through annotations in the general case, which is, to our knowledge, novel in implementations of AARA. Using our system, resources analyses can be decomposed into independent phases: a program is compiled into the machine according to its CBPV semantics, it is then automatically rewritten with our effect and typechecked. Finally, the domain-specific constraints obtained by typechecking are solved using arithmetic solvers. Note that this last step does not involve the original programs or the semantics of the programming language, as opposed to previous work.

Producing resources bounds has many non-trivial requirements: one must unravel memory aliasing, lifetime of allocations, algorithmic complexity and invariants, etc. When all those analyses interact, it becomes beneficial to use a formalism that puts them all on equal footings, as opposed to one dedicated uniquely to resource analyses. In our experiments, combinations of those analyses are easier implement and verify thanks to factorization, effect system, and core inference procedure.

Plan Section 2 is dedicated to the wider context of amortized static analysis, the AARA method, and its recent formalizations in linear, Call-By-Push-Value

λ-calculus. This ends with a more detailed summary of our technical setup implementing AARA. In section 3, we introduce our generic abstract machine adopting those advances, and follow with its type system in section 4. We then explain how to encode resource analyses for high-level languages as a Call-By-Push-Value effect in our model in section 5. We discuss our implementation and automation of analyses for functional languages in section 6, and describe perspectives for further research in the conclusion section 7.

2 Context and *State of the Art*

We begin by fixing some important notions regarding *amortized algorithm analysis* [16], reviewing the AARA framework, and presenting its most recent instantiations. Amortized algorithm analysis is a method for determining the complexity of some operations on a data structure. Instead of merely accumulating costs, it represent programs and their resources together as a closed system, and characterizes cost as the minimum of total resource allowing the execution to proceed. We then present with the *Automated Amortized Resource Analysis* framework and its recent advances using *Call-By-Push-Value*. Once those concepts are set up, we end the section with our setup for recovering AARA in a generic Call-By-Push-Value system.

2.1 Amortized Analysis

Amortized analysis is an ubiquitous tool for the analysis of algorithms over the entire lifetime of some structure, introduced by Tarjan [16] to determine asymptotic costs, but it applies just as well to quantitative prediction. Foreseeing the rest of the paper, we will represent programs by abstract machines and follow the nomenclature of previous works [1]. States of the abstract machine are *commands* $c \in \mathbf{C}$, and are made up of a *value* $V \in \mathbf{V}$ and an *continuation stack* $S \in \mathbf{S}$ with syntax $c = \langle V \parallel S \rangle$. Semantics are given by deterministic, small-step reduction which will be assumed to terminate throughout the paper. The execution of a program is therefore a finite sequence of commands $c = (c_i)_{i \leq n}$.

Costs A *cost metric* is a function $m : \mathbf{C} \times \mathbf{C} \to \mathbb{Z}$ giving a cost for a transition $c \to c'$. When $m(c, c') \geq 0$ we call the cost a *debit*, and when $m(c, c') < 0$ we call it a *credit*. Those credits do not occur for some costs models like time and energy, which cannot be recouped. Models with credits, like memory or currency, require credits and follow an extra condition, mimicking absence of bankruptcy: all intermediate costs $\sum_{j \leq i} m_j$ must be positive. Figuratively, this means that memory cannot be freed before having been allocated, that currency cannot be spent from an empty account, etc.

For a sequence of deterministic reductions $(c_i)_{i \leq n}$, we write $m_i = m(c_i, c_{i+1})$ the cost of a reduction step, and $m(c)$ the total cost of the reduction sequence. This total cost is the maximum of costs that can be reached at an intermediate state, that is $m(c) = \max_{i \leq n} \sum_{j \leq i} m_j$.

Potential This formalism for costs can be reformulated as a matter of *transfer of resources*, an idea originally put forward by Tarjan [16]. Assume given a fixed, sufficiently high amount of resources P to run a program (pick any $P \geq m(c)$). Each intermediate command c_i has a positive amount of *allocated* resources $q_i = \sum_{j \leq i} m_j$, and a positive amount of *free* resources $p_i = P - q_i$. At the beginning of execution, we have $p_0 = P$ and $q_0 = 0$, and as reduction progresses, we have the two inductive relations $p_i = p_{i+1} + m_i$ and $q_i + m_i = q_{i+1}$. Therefore, $p_i + q_i = p_0 + q_0 = P$ is an invariant of execution: resources are neither created nor lost, but preserved.

Predicting Amortized Costs The "potential" point of view frames the problem of cost analysis as one of invariant search: given $V \in \mathbf{V}$, find some function $f : \mathbf{S} \to \mathbb{N}$ such that $f(S) \geq m(\langle V \parallel S \rangle)$. Specifically, for each type for environment, we define numerical invariants called *parameters* (size, length, height, etc.) which gives a function $\varphi : \mathbf{S} \to \mathbb{N}^k$ taking each environment to its numerical invariant, and call φ a *parameterization* of S. Many such parameterizations exists for the same datum. For example, a tree might be parameterized by its size, its depth and width, or more complex combinatorial data. Then given a parameterization φ of the runtime data, the amortized complexity of V is a function $P_V : \mathbb{N}^k \to \mathbb{N}$ such that $P_V(\varphi(S)) \geq m(\langle V \parallel S \rangle)$.

2.2 Automated Amortized Resource Analysis and *RAML*

Hoffmann's *Automated Amortized Resource Analysis* (introduced in [4], see [6] for a retrospective) is a type-theoretic framework for resource usage inference in functional languages. We give here a short introduction to AARA for non-recuperable costs. Nevertheless, both AARA and our methods support them.

In Hoffmann's work, costs are represented by pairs $p = (p_{\max}, p_{\text{out}})$ with $p_{\max} \geq p_{\text{out}}$, which means "evaluation has a maximal cost of $p_{\max}$ of which p_{out} are still allocated at the end". They are endowed with sequencing written additively. Judgements $\Gamma \vdash^{p}_{p'} e : T$ means that if p resources are free before evaluating e, then p' are available afterward.

Parameters in AARA are linear combinations of specialized parameters called *indices*, which directly represent the number of some pattern in a structure; For example, the base indices of a list l are the binomial coefficients $\binom{len(l)}{k}$ with constant k, which count the number of non-contiguous sublists of l with length k. The weights with which indices are combined are subject to a linear-programming optimization to derive bounds on $p_{\max}$ and p_{out}. AARA doesn't use a linear type system. Instead, source programs must be *syntactically affine*: every variable is used at most once, and explicitly duplicated, which splits its weights.

We show the rule for pairs below, exhibits an important property of AARA typing: it encodes operational semantics of the specific source language within its typing rules. Below for example, the cost k_{pair} is payed from $p + k_{\text{pair}}$, then e is

evaluated, then e', yielding the sequence of potentials $(p+k_{\text{pair}}) \to p \to p' \to p''$.

$$\frac{\Gamma \vdash^{p}_{p'} e : A \qquad \Gamma \vdash^{p'}_{p''} e' : B}{\Gamma \vdash^{p+k_{\text{pair}}}_{p''} (e, e') : A \times B}$$

Instances of AARA cover different complexity classes [7], some aspects of garbage collection [12] for pure functional programming, and some aspects of imperative programming with mutable arrays [9]. Hoffmann et al. have implemented AARA in *Resource-Aware ML* [5] (RAML), a type system for a purely-functional subset of OCaml that infers memory bounds, and supports reasoning the the number of nodes in algebraic datatypes, iteration on lists, deeply-nested pattern matching, and limited form of closures. On those programs, RAML can infer costs for a class of algorithms of polynomial complexity. The key point allowing RAML to precisely bound memory usage of OCaml programs is its compile-time representation of heap pointers, allowing it to be aware of memory aliasing. RAML support high-order programming, if high-order arguments do not change during successive calls to the high-order function that uses them. Our continuation-passing, defunctionalized system allows for such changes to be represented as modification to the argument's evaluation context, with can in theory be tracked using the same tools as data structures.

2.3 dℓPCF and λ_{amor}

On the other end of the spectrum, type systems inspired from program logics can prove complex properties, even for non-terminating programs. Dal Lago & Gaboardi's dℓPCF [3,2] is a type system for the λ-calculus with integers and fixpoints (PCF for short), with a highly-parametric, linear, and dependent type system. It is relatively complete up to a solver for arithmetic. This can encode, for example, the number of execution steps of programs in the Krivine machine in the presence of fixpoints, but finding closed forms of this number of steps is undecidable. This highlights the impossibility of typing the costs of fixpoint in the general case.

Originally, dℓPCF could only bound accumulative costs, but subsequent work by Rajani, Gaboardi, Garg & Hoffmann [15] have recovered amortization within this setting. The resulting system, λ_{amor}, is a family of program logics, parameterized by a first-order theory describing resources. Changing this theory tunes the resulting system to be close to the syntax-directed, inferable, and amortized costs of AARA, or the recursion logic of dℓPCF. Resources and costs are represented using two primitive type constructors. Computation incurring costs are typed $\mathbb{M}_I A$ (with I the cost), and implement a cost-accumulating monad. The type of values of type A holding potential is $[I]A$, and implements the dual potential-spending comonad. This system uses Levy's *Call-By-Push-Value* [8] formalism for encoding effectful λ-calculi, with two apparent limitation: first, it uses two different reductions: one for cost-free expression, and second that performs resource interaction on normal forms of the previous one. Second, the finer semantics of Call-by-Push-Value are only used in λ_{amor} to analyse programs with coarser Call-by-Value semantics.

2.4 Abstract Interpretation for Resource Analysis

While the current work focuses on type-based resource analysis and extension to finer program semantics, other resource analysis techniques have been created. We note the *CioaPP* system for resource analysis [10], based on *abstract interpretation*. This techniques approximates the semantics of values using *abstract domains*. For example, and integer could be abstracted by an union of interval, to which the integer is guaranteed to belong. Multiple languages are supported through compilation to *Horn clauses* in the style of logic programming: programs are represented by predicated $C(\overrightarrow{x})$ linking their source and semantics, defined by relation to other predicates in clauses $C'_1(\overrightarrow{x}) \wedge \cdots \wedge C'_2(\overrightarrow{x}) \Rightarrow C(\overrightarrow{x})$, in which the $\overrightarrow{x}$ are all quantified universally. Abstract domains for $\overrightarrow{x}$ can then be directly built using the clauses, and a purpose-built *fixpoint operator* for each domain that abstract iteration. *Cioa/PP* can be used to derive polynomial, exponential and logarithmic complexities, and verify that programs manipulate resources according to quantitative bounds over all inputs or a restriced domain. Implementations exists for many monotone resource metrics, such as time, energy and *gas* (execution fees for smart contracts on blockchains) [13]. To our knowledge, recuperable resources aren't supported.

2.5 Our Technical Setup

Decomposing AARA into a Call-By-Push-Value effect allows for a simplified embedding of languages and programming paradigms: Call-By-Push-Value programs explicitly define their evaluation order and allow for mixed-style evaluation. We exploit this in the next section. Furthermore, this allows embedding AARA's index languages into a mainstream, general-purpose type system (sequent-style System-F) and simplifies formalization. We'll describe our type system and how to encode resource analyses in sections 4 and 5. As a consequence of those two changes, the vast literature of typechecking and type inference then becomes directly applicable, which we discuss in section 6.

3 The ILL-calculus

We now introduce the *polarized **ILL**-calculus*, an abstract machine calculus due to Curien, Fiore & Munch-Maccagnoni[1,11], which we extend with algebraic datatypes, fixpoints, and explicit sharing. The name is a nod to its type-level semantics which are exactly *polarized **I**ntuitionnistic **L**inear **L**ogic*. At runtime, it is exactly a continuation-passing abstract machine for the *Call-By-Push-Value* λ-calculus. This technical setup allows for a state-passing effects, and an encoding resource manipulations at type-level. The core of the machine is taken as-is from [1], and we introduce the following additions for our purposes: explicit sharing of variables; polymorphism; fixpoints; and a notation for thunks and closures.

3.1 Generalities

At the term level, the **ILL**-machine is an abstract machine, whose terms are described in "**Terms (linear)**", figure 1. The first line defines *commands* c, which are a pair $\langle V \parallel S \rangle$ of a value V and a stack S. Both parts are tagged with a *polarity*: $+$ for data and $-$ for computations. When a value and a stack interact in a command, we call V *left side* and S *right side*.

Below commands in the figure, values and stacks are defined in matching pairs of same polarity. Some values are built inductively from constructors $V = K(\overrightarrow{V})$ and interact with pattern matching stacks with many branches $S = \mu(K(\overrightarrow{x}).c \mid \dots)$. Dually, some stack defined inductively and terminated with a *continuation variable*, giving $S = K_1(\overrightarrow{V_1}) \cdot K_2(\overrightarrow{V_2}) \cdot \dots \cdot \alpha$, and interact with pattern-matching values $V = \mu((K(\overrightarrow{x}) \cdot \alpha).c \mid \dots)$. The continuation variable α stands for a yet-unspecified stack, and value variables x stand for yet-unspecified values. This implements *continuation passing*: instead of *returning* a value, programs *jump to the current continuation* by passing it to a stack.

3.2 Linear Fragment

The **ILL** machine works with *linear substitutions* unless stated otherwise. In this subsection, the machine encodes linear computations, which preserve resources held by values *by definition*. We now describe each pair of compatible values and stack which use linear substitution. For each following **(bold label)**, please refer to the corresponding definition in **Terms** and reduction in **Reduction**.

(let-value)* and *(let-stack) The machine manipulates values and stacks with binders: the stack $\mu^+ x.c$ captures the data on the other side, and jumps to c with x bound, which can be understood as "`let` $x = \dots$ `in` c". Dually, the term $\mu^- \alpha.c$ captures the evaluation context on the other side of the command in α and jumps to c. Those are the two reductions in **Reduction**, figure 1.

(data) Algebraic type constructors are defined as in functional languages *à la* OCaml or Haskell. They have value-constructors K to build values with the familiar syntax $K(\overrightarrow{V})$. Data structures then are consumed by *pattern-matching* stacks: for example, a type with two constructors $K_1(-)$, and $K_2(-,-)$ matches with a stack $\mu(K_1(x_1).c_1 \mid K_2(x_2, y_2).c_2)$. Those two reduce together by branching and binding variables:

$$\langle K_1(V_1) \parallel \mu(K_1(x_1).c_1 \mid K_2(x_2, y_2).c_2) \rangle \to c_1[V_1/x_1].$$

(computation) The same way datatypes has values build inductively from constructors $K(\overrightarrow{V})$, computation types have *stacks* defined inductively from constructors $K(\overrightarrow{V}) \cdot S$. Functions are the prototypal example: the function $A \multimap B$

Terms (linear)

$$
\begin{array}{rcccccc}
c ::= & \langle V^+ \parallel S^+ \rangle^+ \mid \langle V^- \parallel S^- \rangle^- & & & & & \textit{(cut)} \\
 & & & & & & \\
 & \textit{(var)} & \textit{(let-val)} & & \textit{(data)} & & \textit{(closure)} \\
V^+ ::= & x^+ & \diagup & & K(\overrightarrow{V^+}) & & \Downarrow(V^-) \\
S^+ ::= & \alpha^+ & \mu x^+.c & & \mu\left(\overrightarrow{K(\overrightarrow{x^+}).c}\right) & & \mu\Downarrow(x^-).c \\
 & & & & & & \\
 & \textit{(var)} & \textit{(let-stk)} & & \textit{(computation)} & & \textit{(thunk)} \\
V^- ::= & x^- & \mu\alpha^-.c & & \mu\left(\overrightarrow{(K(\overrightarrow{x^+})\cdot\alpha^-).c}\right) & & \mu(\Uparrow\cdot\alpha^+).c \\
S^- ::= & \alpha^- & \diagup & & K(\overrightarrow{V^+})\cdot S^- & & \Uparrow\cdot S^+
\end{array}
$$

Terms (non-linear)

$$
\begin{array}{rcccc}
c ::= & \langle V^+ \parallel \mu\mathbf{del}.c\rangle \mid \langle V^+ \parallel \mu\mathbf{dup}(x,y).c\rangle & & & \textit{(structure)} \\
 & & & & \\
 & \textit{(sharing)} & & \textit{(fixpoint)} & \\
V^+ ::= & \mu(!\cdot\alpha^-).c & & \mu(\mathbf{fix}\cdot\alpha^-).\langle\mathbf{self} \parallel S^+\rangle & \\
S^+ ::= & !\cdot S^- & & \mathbf{fix}\cdot S^- &
\end{array}
$$

Reductions

$$
\begin{array}{rcccc}
\textit{(let-stack)} & \left\langle \mu^+\alpha.c \,\middle\|\, S \right\rangle & \to & c[S/\alpha] \\
\textit{(let-value)} & \left\langle V \,\middle\|\, \mu^- x.c \right\rangle & \to & c[V/x] \\
\textit{(weakening)} & \left\langle V \,\middle\|\, \mu\mathbf{del}.c \right\rangle & \to & c \\
\textit{(contraction)} & \left\langle V \,\middle\|\, \mu\mathbf{dup}(x,y).c \right\rangle & \to & c[V/x, V/y] \\
\textit{(closure)} & \left\langle \Downarrow(V) \,\middle\|\, \mu\Downarrow(x).c \right\rangle & \to & c[V/x] \\
\textit{(thunk)} & \left\langle \mu(\Uparrow\cdot\alpha).c \,\middle\|\, \Uparrow\cdot S \right\rangle & \to & c[S/\alpha] \\
\textit{(datatypes)} & \left\langle K_j(\overrightarrow{V}) \,\middle\|\, \mu\left(\overrightarrow{K_i(\overrightarrow{x_i}).c_i}\right) \right\rangle & \to & c_j[\overrightarrow{V}/\overrightarrow{x_j}] \\
\textit{(computations)} & \left\langle \mu\left(\overrightarrow{(K_i(\overrightarrow{x_i})\cdot\alpha_i).c_i}\right) \,\middle\|\, K_j(\overrightarrow{V})\cdot S \right\rangle & \to & c_j[\overrightarrow{V}/\overrightarrow{x_j}, S/\alpha_j] \\
\textit{(sharing)} & \left\langle \mu(!\cdot\alpha).c \,\middle\|\, !\cdot S \right\rangle & \to & c[S/\alpha] \\
\textit{(fixpoint)} & \left\langle \mu(\mathbf{fix}\cdot\alpha).\langle\mathbf{self} \parallel S\rangle \,\middle\|\, \mathbf{fix}\cdot S' \right\rangle & & \\
 & & \to & \langle\mu(\mathbf{fix}\cdot\beta)\langle\mathbf{self} \parallel S[\beta/\alpha]\rangle \parallel S\rangle[S'/\alpha]
\end{array}
$$

Fig. 1: **ILL**-machine: term-level syntax

has a constructor $\lambda(V) \cdot (S)$ which carries an argument of type A and a continuation stack consuming A B. Then, the λ-calculus call $f(V)$ corresponds to the command $\langle f \parallel \lambda(V) \cdot S \rangle$, where S is the outer context of the call (hidden in λ-calculus). The body of f is a pattern-matching value: we write $f = \mu(\lambda(x) \cdot \alpha).c$, in which x binds the argument and α binds the continuation. They interact by reducing as:

$$\langle \mu(\lambda(x) \cdot \alpha).c \parallel \lambda(V) \cdot S \rangle \to c[V/x, S/\alpha].$$

Types with many stack constructors implement computation with many different calls, each call sharing the same environment (think, in OOP, of an object with multiple methods, all sharing the same instance variables).

(thunks $\Uparrow$)* and *(closures $\Downarrow$) Closures and thunks implement local control flow, by delaying calls to computations and generation of data. Thunks $\mu(\Uparrow \cdot \alpha).c$ are commands c blocked from reducing, with free stack variable α. When evaluated together with a stack $\Uparrow \cdot S$, they bind S to α, and jump to c. Formally, they reduces as $\langle \mu(\Uparrow \cdot \alpha^+).c \parallel \Uparrow \cdot S^+ \rangle \to c[S^+/\alpha^+]$. The commands $c[S/\alpha]$ immediately evaluates the thunk, and eventually its return value will interact with S. Closures go the other way around, and delay launching computation. This allows, for example, to store a function within a data structure. They are the symmetric of thunks: a closed computation $\Downarrow(V^-)$ is opened with a blocked context $\mu\Downarrow(x^-).c$, which captures V^- as x^- and launches c, which sets up a new evaluation context for it.

3.3 Call-By-Value Semantics

The canonical encoding of linear call-by-value functions $A \multimap B$ into Call-By-Push-Value translates them as $\Downarrow(A \multimap \Uparrow B)$. At the term-level, the linear function $\lambda x.e$ becomes a closure $\Downarrow$ over a function $\mu(\lambda(x) \cdot \alpha)$, which defines a thunk $\mu(\Uparrow \cdot \beta)$, which evaluates e. We write $[\![-]\!]$ the Call-By-Push-Value embedding of a call-by-value term or type. Putting it all together, we have:

$$[\![A \multimap B]\!] = \Downarrow [\![A]\!] \multimap \Uparrow [\![B]\!]$$

$$[\![\lambda x.e]\!] = \Downarrow \mu(\lambda(x) \cdot \alpha).\langle \mu(\Uparrow \cdot \beta).\langle [\![e]\!] \parallel \beta \rangle \parallel \alpha \rangle$$

The main point of interest of those semantics of CBV in CBPV is that they are extendable with effects, which are implemented as systematic rewriting of thunks and closures. Those rewritings can be sequenced to refine effects. In the last sections, we combine an effect of type-level tracking of quantities and one for state-passing to recover AARA.

3.4 Non-Linear Fragment

In order to encode non-linear programs, including recursion, and track them at type-level, we introduce variations to closures that encode shared values and recursive values.

(sharing) Shareable data is encoded as *shareable commands* $\mu(! \cdot \alpha).c$, whichare shared as-is then pass data to a stack $! \cdot S$. Linearity is enforced at the type level by making them have a distinct type, and asking that all value-variables bound in shared commands also have a shareable type. In order to track non-linear substitutions, sharing is explicitly implemented as stack matching shared values. The stack $\mu\mathbf{del}.c$ implements weakening by silently ignoring the value it matches on, and $\mu\mathbf{dup}(x, y).c$ implements contraction by binding two copies x and y of the shared value.

(fixpoint) Finally, recursive computations are encoded as *fixpoints*. They are also subject to weakening and contraction, which enables the usual recursion schemes of λ-calculus to be encoded into **ILL**. Formally, the stack constructor $\mathbf{fix} \cdot S$ opens a fixpoint closure, expands its definition in its body once, and feeds the resulting computation to S. On the other side, fixpoints have syntax $\mu(\mathbf{fix} \cdot \alpha).\langle \mathbf{self} \parallel S \rangle$, where **self** is a hole to be filled with the recursive value, and S captures the self-referent closure once filled-it and returns the defined recursive computation to α. The reduction substitutes the entire fixpoint into **self**, which copies S. This is made formal in the associated rule in **Reduction**, figure 1. Note the α-conversion in this rule, which protects one of the copies of S from unwanted substitution.

4 Type System

The end game of the type system is to derive a first-order constraint C over relevant quantities of a program, from which we then derive a bound. We call those quantities *parameters*. They represent amount of liquid resources, or combinatorial information on data and computation. In this paper, we focus on parameterizing data, for brevity.

It is capital that computations operate on data of arbitrary parameters. For example, fixpoints will call themselves with arguments of varying sizes to encode iteration. This means polymorphism over size *must* be accounted for. We solve this issue by bundling quantified parameters within constructors.

4.1 Generalities

At the type level, **ILL** is polarized *intuitionistic linear sequent calculus*. Its syntax is described in **Types** and **Parameters**, figure 2. The types A, B, C of values and stack can have two base sorts $\mathcal{T} \in \{+, -\}$ reproducing their polarity. We also have parameters n, p, q, with sorts in $\mathcal{P}$ which includes the integers $\mathbb{N}$. Types can depend on parameters: they have sorts $\overrightarrow{\mathcal{P}} \to \mathcal{T}$. Finally, type constructors are polymorphic over types and parameters: they have sort $\overrightarrow{\mathcal{T}} \to \overrightarrow{\mathcal{P}} \to \mathcal{T}$. For example, lists can have heterogeneous parameters for each element: a list $[a_0; a_1; \ldots; a_n]$ which each a_i of type $A(i)$ has type $\mathtt{List}(A, n)$, with the arguments having type $A(n-1)$ for the head, then $A(n-2)$, all the way to $A(0)$. The associated type constructor is $\mathtt{List} : (\mathbb{N} \to +) \to \mathbb{N} \to +$.

Primitives and Parameters The usual connectives of instuitionistic linear logic are definable as type constructors: $\otimes$ and $\mathbb{1}$ ("pattern matching pairs" and "unit type"), $\oplus$ and $\mathbb{0}$ ("sums" and "empty type"), & and $\top$ ("lazy pairs" and "top"), and $\multimap$ ("linear functions"). Thunks and closures are given their own types: closures over a computation A^- have the positive type $\Downarrow A^-$, and thunks returning some data typed A^+ have negative type $\Uparrow A^+$. We also take as given the integers $(\mathbb{N}, 0, 1, +, \times)$ for parameterization. Those can be extended to any first-order signature.

Judgements Judgements are sequents $\Theta; C; \Gamma \vdash \Delta$, which represent a typed interface: inputs are denoted by value variables $\Gamma = (\overrightarrow{x : A})$, and output by one stack variable $\Delta = (\alpha : A)$. The parameters of this interface are $\Theta = (\overrightarrow{p : \mathcal{P}})$ and are guarded by a first-order constraint C. The parameters in Θ are bound in C, Γ and Δ, and denote free quantities than be tuned within limits given by C. Given this, the three judgements, for values V, stacks S and commands c are:

Syntax	*Sequent*	*Given $\boldsymbol{\Theta}$ such that $\boldsymbol{C}$, we have ...*
Values	$\Theta; C; \Gamma \vdash V : A$	*a value $\boldsymbol{V}$ of type $\boldsymbol{A}$ in context $\boldsymbol{\Gamma}$.*
Stacks	$\Theta; C; \Gamma \mid S : A \vdash \Delta$	*a stack $\boldsymbol{S}$ of type $\boldsymbol{A}$ in context $\boldsymbol{\Gamma}$, $\boldsymbol{\Delta}$.*
Commands	$c : (\Theta; C; \Gamma \vdash \Delta)$	*a valid command $\boldsymbol{c}$ under context $\boldsymbol{\Gamma}$, $\boldsymbol{\Delta}$.*

The central rules of the type system are shown in "**Example rules**" figure 2. Commands are built in the (cut) rule by matching a value and stack on their type and taking a conjunction of their constraints. Rules (μL) and (μR) are for binders. For example, (μR) turns a command c with a free variable $x : A$ into a stack $\mu x.c : A$ which interacts with values of type A by substituting them for x.

4.2 Datatypes with Parameters

Building up the logical constraint for resources and algorithmic invariants is done by accumulating generic information about constructors of values and stack. To present how this machinery works, we encode a very simple constraint: *a list is always one element longer than its tail.* The corresponding type definition for lists, and resulting rules are shown in **Example type definition**, figure 2. Lists have type $\texttt{List}(A, n)$ with a type parameter $n : \mathbb{N}$ denoting size. The definition of the `Cons` constructor is reproduced below.

$$\mid \text{ Cons } \textbf{ of } \text{A(m)} \otimes \texttt{List}\,\text{(A,m)} \textbf{ where } (m : \mathbb{N}) \textbf{ with } n = m + 1$$

The definition states that lists $\texttt{List}(A, n)$ have a head of type $A(m)$ and a tail of type $\texttt{List}(A, m)$. The type of list elements is $A : \mathbb{N} \to +$, which allows each element to be given a distinct parameterization according to its position. For example, $\texttt{List}(\texttt{List}(\mathbb{N}, -), n)$ is a type of lists of lists of integer of decreasing lengths: the first list has size 10, the next one 9, etc. Formally, the parameter m is introduced in the **where** clause, and is guarded by the first-order constraint $n =$

Types

$$\mathcal{T} ::= + \mid - \mid \mathcal{P} \mid \mathcal{P} \to \mathcal{T}$$
$$A ::= p \mid T_{\text{cons}}(\overrightarrow{A}) \mid \Downarrow A \mid \Uparrow A \mid !A \mid \mathbf{Fix}\ A$$
$$A ::= \mathbb{1} \mid A \otimes A \mid \mathbb{0} \mid A \oplus A \mid \top \mid A \,\&\, A \mid A \multimap A \mid \exists \Theta[C].C \mid \forall \Theta[C].A \quad \textit{(definable)}$$

Parameters

$$\mathcal{P} ::= \mathbb{N} \mid \ldots$$
$$p ::= 0 \mid 1 \mid p + q \mid \ldots$$
$$C ::= \mathbb{T} \mid \mathcal{R}(\overrightarrow{p}) \mid p = p \mid C \wedge C \mid C \Rightarrow C \mid \exists \overrightarrow{p}.C \mid \forall \overrightarrow{p}.C$$

Example type definition: lists

```
data List(A:ℕ → +, n:ℕ) =
  | Cons of A(m) ⊗ List(A,m) where (m : ℕ) with n = m + 1
  | Nil with n = 0 . (*no 'where' clause for Nil*)
```

Example type definition: state token

```
data ST(p,q : ℕ) =
| init where q = 0
| debit_k  of ST(p',q') with p',q' : ℕ   where p' + k = p ∧ q + k = q'
| credit_k of ST(p',q') with p',q' : ℕ   where p + k = p' ∧ q' + k = q
| slack    of ST(p',q') with p',q',k : ℕ where p' + k = p ∧ q + k = q'
```

Example rules: identity and lists

$$\frac{c : (\Theta; C; \Gamma \vdash \alpha : A^-)}{\Theta; C; \Gamma \vdash \mu^- \alpha.c : A^-}\ (\mu\text{L}) \qquad \frac{c : (\Theta; C; \Gamma, x : A^+ \vdash \Delta)}{\Theta; C; \Gamma \mid \mu^+ x.c : A^+ \vdash \Delta}\ (\mu\text{R})$$

$$\frac{\Theta; C; \Gamma \vdash V : A^{\pm} \qquad \Theta'; C'; \Gamma' \mid S : A^{\pm} \vdash \Delta}{\langle t \parallel e \rangle^{\pm} : (\Theta, \Theta'; C \wedge C'; \Gamma, \Gamma' \vdash \Delta)}\ (\text{cut})$$

$$\frac{\Theta_1; C_1; \Gamma_1 \vdash V_1 : A(m) \qquad \Theta_2; C_2; \Gamma_2 \vdash V_2 : \texttt{List}(A, m)}{\Theta_1, \Theta_2; C; \Gamma_1, \Gamma_2 \vdash \texttt{Cons}(V_1, V_2) : \texttt{List}(A, n)}\ (\texttt{ConsR})$$
$$C = \exists m.(n = m + 1) \wedge C_1 \wedge C_2$$

$$\frac{c_1 : (\Theta; C_1; \Gamma \vdash \Delta) \qquad c_2 : (\Theta, m : \mathbb{N}; C_2; \Gamma, x : A(m), y : \texttt{List}(A, m) \vdash \Delta)}{\Theta; C; \Gamma \mid \mu(\ \texttt{Nil}().c_1 \mid \texttt{Cons}(x, y).c_2\ ) : \texttt{List}(A, n) \vdash \Delta}$$
$$C = ((n = 0) \Rightarrow C_1) \wedge (\forall m.(n = m + 1) \Rightarrow C_2) \quad (\texttt{ListL})$$

Fig. 2: **ILL-machine: type system and examples**

$m+1$ in the **with** clause. This is to be understood as "**where** m is fresh integer parameter **with** $n = m + 1$". When constructing a list $\texttt{List}(A, n)$ with the rule ($\texttt{Cons}$R), m is added to Θ and the constraint $n = m+1$ is added. Symmetrically, when pattern matching on a list $\texttt{List}(A, n)$ with the rule ($\texttt{List}$L), the branch of the pattern matching for `Cons` must be well-typed for any m such that $n = m+1$, which yields a constraint $\forall m.(n = m + 1) \Rightarrow C'$ in which $n = m + 1$ is assumed.

4.3 Implementing Polymorphism

Once some parameterization of data is chosen, we want parameters-aware data and computations. Recall that when introducing constructors, some parameters Θ satisfying some constraint C are introduced existentially, and when this constructor is matched on, they are introduced universally. This allows us to encode polymorphism over parameters as types with only one constructor. We define existential quantification over parameters Θ such that C with the $\exists\Theta[C].A$ datatype as follows:

$$\textbf{data}\ \exists\Theta[C]\,.\mathrm{A} = \texttt{Pack}_{\Theta;C}\ \textbf{of}\ \mathrm{A}(\Theta)\ \textbf{where}\ \Theta\ \textbf{with}\ C$$

Introducing $\texttt{Pack}_{\Theta;C}(V)$ produces the constraint $\exists\Theta.C$, and when pattern-matching on it, C is assumed to hold for some unknown Θ, giving a constraint $\forall\Theta.C \Rightarrow C''$. Universal quantification goes the other way around, binding some constraint existentially in stacks, and universally in values. Compile-time information Θ, C about a continuation stack $S : A^-(\Theta)$ is witnessed by the stack constructor $\texttt{Spec}_{\Theta;C} \cdot S$. On the value side, $\mu(\texttt{Spec}_{\Theta;C} \cdot \alpha).c$ requires that C in the command c, and generates $\forall\Theta.C \Rightarrow C'$.

Closures over universally quantified computations take any input such that some constraint holds. Likewise, thunks over existential quantification type delayed computations with (yet undetermined) parameters. For example, a thunk which returns a pair of lists whose total length is 10 can be typed as $\Uparrow\exists(n, m : \mathbb{N})[n + m = 10].\ \texttt{List}(A, n) \otimes \texttt{List}(A, m)$.

4.4 An Example of Encoding: *append*

A minimal, non-trivial example of parameter polymorphism is the `append` function on lists. We implement it in two phases. First, we implement `rev_append`, which flips the first lists and appends it to the second. Then, we define `append` with two calls to `rev_append`. This decomposition shows the function of parameter polymorphism, as both call to `rev_append` occur on lists of different sizes. Figure 3 lists the original ML code for both functions and the compiled version of `rev_append`. We omit giving the translation of `append`, since it is straightforward once given `rev_append`.

`rev_append` is defined as a fixpoint. The first line in the definition binds a self-reference to f and binds a continuation α to which it returns the function. In c_1, when the function is called for lists l_1 of size n and l_2 of size m, the execution context built by the caller instantiates the sizes with $\forall_{(n,m)}$ which the callee matches on. Then, in c_2, we pattern-match on l_1, which introduces

```
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
```

$$\texttt{rev\_append} : \ \mathbf{Fix}\ \forall n, m, A.\ \mathtt{L}(A, n) \multimap \mathtt{L}(A, m) \multimap \Uparrow \mathtt{L}(A, n + m)$$
$$= \mu(\mathbf{fix} \cdot \alpha).\langle \mathbf{self} \parallel \mu f.c_1 \rangle$$
$$c_1 = \langle \mu(\forall_{(n,m)} \cdot l_1 \cdot l_2 \cdot \Uparrow \cdot \beta).c_2 \parallel \alpha \rangle$$
$$c_2 = \langle l_1 \parallel \mu(\ \mathtt{Nil}_{(n=0)}.\langle l_2 \parallel \beta \rangle \mid \mathtt{Cons}_{(\exists n'.n=n'+1)}(h, t).c_3\ )\rangle$$
$$c_3 = \langle f \parallel \mathbf{fix} \cdot \forall_{(n',m+1)} \cdot t \cdot \mathtt{Cons}_{(m+1)}(h, l_2) \cdot \Uparrow \cdot \beta \rangle$$

Fig. 3: **BILL** Source code of the `rev_append` function

$\exists n'.n = n' + 1$ (resp. $n = 0$) if the list has a head (resp. is empty). In this last case, the function recurses on itself in c_3. This recursive call is done with an execution context $\forall_{(n',m+1)} \cdot S$, in which the new values for n and m are instantiated.

4.5 Soundness

Reduction preserves parameterizations in the following sense:

Theorem 1. *If $c : (\Theta; C; \Gamma \vdash \Delta)$ reduces as $c \to c'$, and $c' : (\Theta'; C'; \Gamma' \vdash \Delta')$, then $\Theta' \subset \Theta$, $\Gamma' \subset \Gamma$, $\Delta = \Delta'$, and for every instantiation of Θ, $C \Rightarrow C'$.*

The proof is done by induction over typed reduction of commands, after proving the standard Barendregt properties (which can be done following [11]). We only have space to briefly summarize the salient point. First, we prove the statement for $\langle \mu\alpha.c \parallel S \rangle$ and $\langle V \parallel \mu x.c \rangle$. Then, the only significant cases are $\langle \mathtt{Pack}_{\Theta;C}(V) \parallel \mu \mathtt{Pack}_{\Theta;C}(x).c \rangle$ and its dual with `Spec`. The `Pack` command reduce to $c[V/x]$ and generates the constraint $(\exists \Theta.C) \wedge (\forall \Theta.C \Rightarrow C')$ with C' the constraint for c. This immediately implies $C \wedge C'$, which is also constraint generated by $\langle V \parallel \mu x.c \rangle$, and therefore $c[V/x]$. The case of `Spec` is purely identical.

5 Embedding AARA as an Effect

In section 3, we refined the CBPV embedding of CBV functions to polymorphic closures and thunks. This allowed to track parameters as control flow switches in and out of programs. To recover AARA, we merely need to specialize this translation to track sizes and resources.

5.1 An Effect for Parameters

With our setup, we can translate CBV programs to **ILL**-machine that associates a constraint C on their free parameters Θ. This is implemented by refining closures and thunks. Closures $\Downarrow A$ are replaced by *closures over quantified computations* $\Downarrow \forall \Theta[C].A(\Theta)$, that is, computations that take *any* arguments with

parameters Θ satisfying C. On the other side, thunks $\Uparrow B$ are replaced with $\Uparrow\exists\Theta'[C'].B(\Theta')$, which returns data parameterized by Θ' such that C'. With this effect, the call-by-value linear function $A \multimap B$ is translated to a parameter-aware version that accept *all* ($\forall$) inputs with the right parameters and return *some* ($\exists$) output with its own parameters. For example, the function `append` on lists has a length-aware type (here in long form, to show the implicit $\exists$ binder):

$$\Downarrow\forall n, m.\ \texttt{List}(A, n) \multimap \texttt{List}(A, m) \multimap \Uparrow\exists k[k = m + n].\ \texttt{List}(A, k)$$

5.2 Polymorphic State-Passing Effect

We extend the translation of call-by-value functions with another effect: state passing. Closures now accept a token $\mathrm{ST}(p, q)$ with p free resources and q allocated resources, and thunks return a token $\mathrm{ST}(p', q')$. Closures $\Downarrow\forall\Theta[C].(-)$ become $\Downarrow\forall\Theta[C].\mathrm{ST}(p, q) \multimap (-)$, in which C guards the resources p and q. This means closures take in *any* state whose resources satisfy C. Likewise, thunks becomes $\Uparrow\exists\Theta[C'].\mathrm{ST}(p', q') \otimes (-)$, and return resources p' and q' specified by C'.

This can lift the hidden inner behavior of `append` at type-level. Relying on linear typing, the effects detect that the progressive deallocations of the intermediate reversed list compensate for the progressive allocation of the final result (under an ideal garbage collector). When typed with state-passing below, the tokens' types shows that only n nodes are allocated *simultaneously* for the call.

$$\begin{aligned}\Downarrow\forall n, m, p, q.\ \mathrm{ST}(p + n * k_{\texttt{Cons}}, q) \multimap \texttt{List}(A, n) \multimap \texttt{List}(A, m)\\ \multimap \Uparrow\exists k[k = m + n].\ \mathrm{ST}(p, q + n * k_{\texttt{Cons}}) \otimes \texttt{List}(A, k)\end{aligned}$$

5.3 Token Encoding

To implement this translation without specific primitives, we define a state token capable of representing the operations of *debit* (spending resources), *credit* (recovering resources), and *slack* (aligning costs upwards) at type-level. We define a type constructor ST with two resource parameters, that implements those operations, in **Example type definition: state token**, figure 2.

The token begins its life as `init`, which has type $\mathrm{ST}(p, 0)$. Debiting k_0 resources from a token $s : \mathrm{ST}(p+k_0, q)$ is done by using the constructor $\texttt{debit}_{k_0}(s)$ which creates a new token of type $ST(p, q + k_0)$. The `credit` constructor implements the opposite operation. For slack, the amount of resources being wasted k is left free. At call site, k is introduced existentially in the constraint, and left to be specified later at the whims of the constraint solver. Lower values of k lead to better bounds, but must remain high enough to run all branches.

5.4 Potential in Shared Values

AARA usually stores potential within shared values as opposed to a centralized token. We take a somewhat different approach to sharing: we want shared values

$!A(\Theta)$ to specify how much potential their instances commonly occupy, and store those resources in the token. To do so, we define below a type $!_\varphi A(\Theta)$ which represent shared copies with potential. Building the value requires a token with $\varphi(\Theta)$ free resources and allocates them; Extracting the underlying $!A(\Theta)$ from $!_\varphi A(\Theta)$ frees $\varphi(\Theta)$ resources. This is definable without any primitive in the **ILL**-machine, together with resource-aware contraction and weakening, which fully reproduce AARA potential.

$$!_\varphi A(\Theta) = \Downarrow \forall p, q.\ \mathrm{ST}(p, q + \varphi(\Theta)) \multimap \Uparrow \mathrm{ST}(p + \varphi(n), q) \otimes\ ! \Uparrow A(\Theta).$$

6 Implementation for ML-like Languages

We have implemented a prototype[1] of the **ILL**-machine, together with type inference and constraint generation using the HM(X) technique [14]. HM(X) (*Hindley-Milner extended with X*), is a generic constraint-based type inference procedure extending Hindley-Milner with user-definable sorts, types and predicates. Our extension of covers arbitrary first-order signatures for parameters, and features a generic constraint simplifier. It then can export simplified constraints to the *Coq* theorem prover for verification by hand, or to the *MiniZinc*[2] optimization suite to for full cost inference with minized slack. This yields complexity bounds as a multivariate polynomial. Our heuristic for elaborating polynomial parameter expressions from a first-order constraint works as following:

1. Take as input a first-order constraint over the integers. Its syntax is generated by $(\forall, \exists, \wedge, \Rightarrow, =, \leq)$
2. Skolemize all existential variables: $\exists y.C$ generates a fresh multivariate polynomial $p(\overrightarrow{x})$ over the variables $\overrightarrow{x}$ in scope, and reduces to $C[p(\overrightarrow{x})/y]$. Those polynomial p are formal sums of monomials with coefficients $\overrightarrow{\alpha}$, which are all held in a global context for the polynomials.
3. Assume all implications are of the form $(p(\overrightarrow{x}) = e[\overrightarrow{x}]) \Rightarrow C$, and substitute p for e in C
4. Put the constraint under prenex form. We have arrived at a constraint $C = \forall \overrightarrow{x}.\ \bigwedge_i (e_i[\overrightarrow{x}, \overrightarrow{\alpha}] = e'_i[\overrightarrow{x}, \overrightarrow{\alpha}])$. Reinterpret the constraint as a system of polynomial equations with variables $\overrightarrow{x}$ and unknown coefficients $\overrightarrow{\alpha}$.
5. Finally, instantiate and optimize $\overrightarrow{\alpha}$ under this final system of equations. The metric for the optimization is the sum of the leading non-zero coefficients of the complexity being computed.

Our preliminary experiments indicate that when manually annotating datatypes definitions with their RAML parameterization, the tight algorithmic complexities derivable for list iterators can be recovered in **BILL**. We are currently exploring ways to extend parameterization to tree-shaped datatypes, as well as the potential precision gains that can be obtained by parameterizing to Call-By-Push-Value evaluation contexts.

[1] https://gitlab.lip6.fr/suzanneh/autobill
[2] https://www.minizinc.org

The direct translation from ML-style languages to **ILL**-with-token is factored in the implementation. User-facing languages are be translated to a Call-By-Push-Value λ-calculus the canonical way, and then compiled into **ILL** through a CPS-translation and explicit duplication of shared variables. Later passes implement the Call-By-Push-Value effects described in this paper, to-and-from **ILL**. High-level languages can be analyzed more easily, as they only need to be translated to a Call-By-Push-Value λ-calculus with credit/debit primitives.

Limitations Higher-order functions are a pain-point for AARA analyses, as getting correct bounds require lifting constraints out of high-order arguments. Depending on the theory modeling parameters, this can be quite tricky. Our implementation compares favorably to RAML in this regard, as it supports native constraints annotations on high-order arguments. Fixpoints are also a thorny issue. This has two mitigations: (1) require user-provided annotations for fixpoint invariants, or (2) reproduce RAML's constraints on iteration: (mutually) recursive functions which define folds and traversals through nested pattern-matching and accumulation. In our experiments, we found our system to be amenable to a third approach: general purpose iterators can be defined with manual annotations using fixpoints, and then used without annotations.

7 Conclusion

Extending the static type discipline of functional languages for resource analysis is a tantalizing prospect. But understanding the operational properties of programs means recovering a diverse swath of information like memory aliasing, algorithmic invariants, and sharing of data outside their scope of definition.

To create a fine-grained, generic, extendable base for AARA, we extended the **ILL** Call-By-Push-Value calculus developed by Curien at al. [11,1] with fixpoints, polymorphism and native first-order constraints. We combined this with a decomposition of λ-calculi in a Call-By-Push-Value machine, which explicits control flow. With this, expressions are a combination of closures requiring some properties to hold on their inputs, and thunks which witness some properties of their output. Instantiating our generic system to represent a finite amount of resources within the program's state, well-typedness stipulates that this finite amount is sufficient to cover all allocations and liberations. It recovers the core of the AARA [6] method for resource analysis from first principles, from a generic Call-By-Push-Value intermediate representation for static analysis.

Perspectives Our implementation covers the target machine, the constraint-aware type system, and a heuristic to solve constraints over multivariate polynomial. Current work focuses on implementing program-wide analysis *à la* RAML. This requires automatically annotating the constraints associated to each constructor in datatype definitions to encode a particular flavor of AARA analyses. We also aim to support shared regions, reusing the parameterized type system to generate constraints on region lifetimes.

References

1. Curien, P.L., Fiore, M., Munch-Maccagnoni, G.: A theory of effects and resources: Adjunction models and polarised calculi. In: Proceedings of the ACM on Programming Languages (POPL) (Jan 2016). https://doi.org/10.1145/2837614.2837652
2. Dal Lago, U., Gaboardi, M.: Linear Dependent Types and Relative Completeness. In: 2011 IEEE 26th Annual Symposium on Logic in Computer Science (Jun 2011). https://doi.org/10.1109/LICS.2011.22
3. Dal lago, U., Petit, B.: The Geometry of Types. ACM SIGPLAN Notices (POPL) (Jan 2013). https://doi.org/10.1145/2480359.2429090
4. Hoffmann, J.: Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis. Ph.D. thesis, Ludwig-Maximilians-Universität München, Berlin (2011), https://doi.org/10.5282/edoc.13955
5. Hoffmann, J., Das, A., Weng, S.C.: Towards automatic resource bound analysis for OCaml. In: Proceedings of the ACM on Programming Languages (POPL) (2017). https://doi.org/10.1145/3009837.3009842
6. Hoffmann, J., Jost, S.: Two decades of automatic amortized resource analysis. Mathematical Structures in Computer Science pp. 1–31 (Mar 2022). https://doi.org/10.1017/S0960129521000487
7. Kahn, D.M., Hoffmann, J.: Exponential automatic amortized resource analysis. In: International Conference on Foundations of Software Science and Computation Structures. pp. 359–380. Springer, Cham (2020)
8. Levy, P.B.: Call-By-Push-Value. Springer Netherlands, Dordrecht (2003). https://doi.org/10.1007/978-94-007-0954-6
9. Lichtman, B., Hoffmann, J.: Arrays and References in Resource Aware ML. In: Miller, D. (ed.) International Conference on Formal Structures for Computation and Deduction (FSCD) (2017). https://doi.org/10.4230/LIPIcs.FSCD.2017.26
10. Lopez-Garcia, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., Hermenegildo, M.V.: Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. Theory and Practice of Logic Programming **18**(2), 167–223 (Mar 2018). https://doi.org/10.1017/S1471068418000042
11. Munch-Maccagnoni, G.: Note on curry's style for linear call-by-push-value. Tech. rep. (May 2017), https://hal.inria.fr/hal-01528857
12. Niu, Y., Hoffmann, J.: 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. https://doi.org/10.29007/xkwx
13. Pérez, V., Klemen, M., López-García, P., Morales, J.F., Hermenegildo, M.: Cost Analysis of Smart Contracts Via Parametric Resource Analysis. In: Pichardie, D., Sighireanu, M. (eds.) Static Analysis, vol. 12389, pp. 7–31. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-65474-0_2
14. Pottier, F., Didier Rémy: The essence of ML type inference. In: Pierce, Benjamin C (ed.) Advanced Topics in Types and Programming Languages, pp. 389–489. The MIT Press (Jan 2005)
15. Rajani, V., Gaboardi, M., Garg, D., Hoffmann, J.: A unifying type-theory for higher-order (amortized) cost analysis. Proceedings of the ACM on Programming Languages (POPL) (Jan 2021). https://doi.org/10.1145/3434308
16. Tarjan, R.E.: Amortized Computational Complexity. SIAM Journal on Algebraic Discrete Methods (Apr 1985). https://doi.org/10.1137/0606031