UNIVERSITÉ PARIS 13 — INSTITUT GALILÉE

THESIS
To obtain the title of
DOCTOR OF SCIENCE OF UNIVERSITÉ PARIS 13
Specialty: Computer Science

GNU epsilon
an extensible programming language

Presented by **Luca Saiu** to be defended in public on November 19th 2012

Jury:

Reviewers:	Emmanuel Chailloux	Université Pierre et Marie Curie – Paris 6
	Michel Mauny	ENSTA ParisTech
Advisors:	Christophe Fouqueré	LIPN, Université Paris 13
	Jean-Vincent Loddo	LIPN, Université Paris 13
Examiners:	Roberto Di Cosmo	PPS, Université Paris Diderot – Paris 7
	Manuel Serrano	INRIA Sophia-Antipolis
	Basile Starynkevitch	CEA LIST/DILS
	Peter Van Roy	Université catholique de Louvain, Belgium

Titre français : GNU epsilon — un langage de programmation extensible

Laboratoire d'Informatique de Paris-Nord, UMR 7030 — CNRS, Univ. Paris 13

Abstract

Reductionism is a viable strategy for designing and implementing practical programming languages, leading to solutions which are easier to extend, experiment with and formally analyze.

We formally specify and implement an extensible programming language, based on a *minimalistic first-order imperative core language* plus strong *abstraction mechanisms*, *reflection* and *self-modification* features. The language can be extended to very high levels: by using Lisp-style *macros* and code-to-code *transforms* which automatically rewrite high-level expressions into core forms, we define closures and first-class continuations on top of the core.

Non-self-modifying programs can be analyzed and formally reasoned upon, thanks to the language simple semantics. We formally develop a *static analysis* and prove a *soundness property* with respect to the dynamic semantics.

We develop a *parallel garbage collector* suitable to multi-core machines to permit efficient execution of parallel programs.

Keywords: programming, language, extensibility, macro, transformation, reflection, bootstrap, interpretation, compilation, parallelism, concurrency, garbage collection

Résumé

Le *réductionnisme* est une technique réaliste de conception et implantation de vrais langages de programmation, et conduit à des solutions plus faciles à étendre, expérimenter et analyser.

Nous spécifions formellement et implantons un langage de programmation extensible, basé sur un *langage-noyau minimaliste impératif du premier ordre*, équipé de *mécanismes d'abstraction* forts et avec des possibilités de *réflexion* et *auto-modification*. Le langage peut être étendu à des niveaux très hauts : en utilisant des *macros* à la Lisp et des *transformations de code à code* réécrivant les expressions étendues en expressions-noyau, nous définissons les clôtures et les continuations de première classe au dessus du noyau.

Les programmes qui ne s'auto-modifient pas peuvent être analysés formellement, grâce à la simplicité de la sémantique. Nous développons formellement un exemple d'*analyse statique* et nous prouvons une *propriété de soundness* par apport à la sémantique dynamique.

Nous développons un *ramasse-miettes parallèle* qui convient aux machines multi-cœurs, pour permettre l'exécution efficace de programmes parallèles.

Mots-clés : programmation, langage, extensibilité, macro, transformation, reflection, *bootstrap*, interprétation, compilation, parallélisme, concurrence, ramasse-miettes

A large, crowded maze of a building that is just one part of one branch of the local administration, in the Paris neighborhood. Under the Summer heat I've been standing there or somewhere very close since the early morning, awake since before 6am just for the privilege of being near the front of the line. It's finally my turn, after half a day spent waiting. And now she tells me that no, my *avis d'imposition fiscal* is not a valid *justificatif de domicile*. And who cares if they had told me the opposite in her very office: she has no intention of listening to my complaints. I'll have to return another day, with a signed copy of my landlord's identity card.

After I get back to the main hall near the entrance to arrange that next appointment I must look as irritated as I am. The woman at the desk asks me what happened. When I repeat to her what I've been told just a few minutes before, she explodes. —*What!? Come with me.* Shouting that she'll be right back to the people waiting behind me, she abandons her place and angrily storms away to another office. I follow her.

We sit. Between an half-muttered insult to her colleague and the next she asks me for my papers one by one, and checks each of them. She has to come back to her own work: fueled by adrenaline she's thorough but efficient. Last, I hand her my *avis d'imposition fiscal*. She compares the address, looks at the date, and skims the rest. —*Yes, it's perfectly fine! That*— labeling her colleague by a final one-word definition. She signs my dossier herself, overriding or simply ignoring the other's authority. I'll have to go pay the tax at the cash register, yes, right there on the left and yes, then I'm done. I barely have the time to thank her before she runs back to her desk.

To that blondish, forty-something woman who was working in a government building near Paris during the Summer of 2010, whatever her name is, I dedicate this work.

May her rage inspire others to do the right thing.

Luca Saiu, December 2012

Acknowledgments

At the beginning of it all, one day of March, I left the place where I was born, alone on my already old car loaded with everything I owned, headed for Paris. Not speaking the language at the time I arrived in France as an awkward foreigner depending on others' kindness. Luckily many people were indeed kind to me.

And soon enough I fell in love with France: to my eyes it looked like everything a good country should be. Yet, every time I expressed this enthusiasm (in English) to my first French friends, I was gently reminded that my vision was partial, my experience too limited. In other words I was a naïf simpleton who hadn't understood anything yet — they said it more softly, *ça va sans dire*, but the idea was clear.

My optimistic naïveté, the will of trusting strangers *because it's the right thing* is more of a philosophical choice of mine — which is to say, isn't actually real. But for the rest I have to admit that those people, advising me to be more cautious in my judgements, were not wrong.

Years have since passed. I met many people in France, all across the spectrum from the constructively angry civil servant of the dedication to her opponent *the lady of many nicknames*. Of everybody I met here I'd say that more people were closer to the former. Everything considered, I'm grateful for the opportunities I got at Université Paris 13.

Thanks to Jean-Vincent Loddo, who trusted me and made all of this possible by hiring me to work on Marionnet during my first six months in France, and then co-directed my doctorate. Jean-Vincent has also been a friend, very reliable and always patient, knowing when to insist and when to let me be. I've never been able to approach my other thesis co-director, Christophe Fouqueré, quite as closely from a human point of view. This is entirely my fault, since Christophe has shown the same good qualities and has been just as helpful and understanding as Jean-Vincent whenever I've asked; he even spontaneously offered me concrete help when I was dealing with practical problems, such as the first time my car got broken-in and I had to go to the police, still not speaking French — and then there are the other episodes he remembers. To both Christophe and Jean-Vincent, thanks. Thanks for real.

I particularly wish to thank some other people in the lab for their friendliness and warmth: Laura Giambruno, Sébastien Guérif, Pierre Boudes (who also offered to help in the same break-in case, and not only then), and Franck Butelle. Christian Codognet, Laure Petrucci and Adrian Tanasa also deserve a high place in this list. And Sophie Toulouse, and Hayat Cheballah.

I'll miss the many roommates I had in office A207 during these years: Jean-Vincent, and then in particular Virgile Mogbil, Daniele Terreni and Giulio Manzonetto. And Sylviane Schwer.

I had an especially good relationship with the other people working on the Marionnet project: of course Jean-Vincent, and then especially Jonathan Roudiere, Marco Stronati, Abdelouahed, and Franck Butelle.

Sébastien Guérif is a friend and possibly the best colleague I've ever had: meticulous, very hardworking, interested in feedback. I've learned a lot about teaching while doing class and lab exercises for his *Programmation Impérative* course. I also enjoyed the company of the other colleagues doing exercises for Sébastien's courses, particularly Daniel Kayser, Christophe Tollu, Antoine Rozenknop, Hanane Allaoua, Manisha Pujari, and again Laura Giambruno.

Thanks to Jean-Yves Moyen as well, for approving more or less every idea I had in organizing the *Programmation Fonctionnelle Avancée* course for two years, implicitly accepting to scale down his role to just lab exercises, despite his experience. It was fun.

Some students of mine were actually interested in what I had to say. I wish to thank them for that sparkle in their eyes saying "that's cool" at the same time as "I get it". Thanks to my bad students as well, if they really tried — but no thanks at all to cheaters.

I fondly remember a couple idle afternoons spent with Christine Recanati speaking about philosophy. I had other agreeable exchanges, geeky, intellectual or simply human, with James Avery, Roberto Wolfler Calvo, Jean-Christophe Dubacq, Jalila Sadki Fenzar, Mario Valencia Pabon, Marco Pedicini. Thanks to Micaela Mayero, Patrick Baillot, Damiano Mazza, Érick Alphonse and Yue Ma as well.

Thanks to Morena Olivieri, who's a good person but doesn't believe it yet. Thanks for setting me in a good enough mood to finally decide to quit smoking. I can see in retrospect how stressful quitting was; Morena and many others encouraged and supported me during that time, particularly José Marchesi and Matteo Golfarini.

I quit in 2010 but of course I still like many smokers, or people who were smokers at the time. Thanks to the friends and colleagues at the lab who helped me socialize and learn the language when meeting outside for a cigarette and some human companionship: first of all Sophie Toulouse and Hayat Cheballah, but also Jonathan Roudiere (again), Jonathan Van Puymbrouck, Pierre Boudes, Amine Hemdane, Vlady Ravelomanana, Frédéric Toumazet, Pascal Coupey, Ferhan Pekergin, Erwan Moreau, Paolo Di Giamberardino, Hichem Kenniche, Haïfa Zargayouna, and the funny guy with a white ponytail who introduced himself as "a mathematician".

Thanks to the old-time friends from the University of Pisa, for the good memories; those who staid, those who escaped like I did, and those who still talk about escaping someday. Thanks in particular to Matteo Golfarini for being a good friend, and also for having me so many times in his place near Madrid; and to my other good friends Carlo Bertolli (who by the way also let me stay in his place in London) and Francesco Nidito.

Some of the others I've not seen in years, but I'm sure they won't feel offended for being included here: Riccardo Vagli, Dario Russo, Marco Righi, Maria Cristina

Favini Berti, Alessio Baldaccini, Antonio Mirarchi, Giandomenico Napolitano, Dimitri Dini, Robert Alfonsi, Massimo Cecchi, Federico Ruzzier, Andrea Venturi, Erika Rossi, Marco Peccianti, Alessio Mazzanti, Sandra Zimei, Marlis Valentini, Eliana Anderlini. And of course Massimiliano Brocchini, who first spoke about me to Jean-Vincent during his time in France.

Then there are some new and found-again friends such as Marco Stronati, Enrico Rubboli and Roberto Pasini — Yet more people who ended up escaping, now that I'm thinking of it.

And Gabriella, who still remembers me. Thanks.

Thanks to Richard Stallman for changing the world by starting the free software movement, for setting an example with his integrity, and for GNU. Several fellow GNU hackers, particularly José Marchesi and Alfred Szmidt, have also become close friends to me; and I wish to remember the former "rabbit" from the UK, who might or might not want to be mentioned by name here.

I remember with pleasure a few beautiful conversations stretching from the evening nearly to the next morning, for example the debate on Algebra we had in Spain with José, Alfred and Aleksander Morgado, going on long after the pub closed and we were sent out at some crazy hour like 4am. Or the long discussions about GCC optimizations and LTO in a Bruxelles hotel hall with Dodji Seketeli, Alfred, José and Laurent Guerby, over what was apparently the only bad beer to be found in Belgium; I had to speak at FOSDEM the next morning — and in the end, after getting barely any sleep, the talk even came out nice.

Of the other GNU people I have to remember at least Nacho Gonzáles, Giuseppe Scrivano, Ludovic Courtès, Andy Wingo, Sylvain Beucler, Bruno Haible (who noticed some surprising similarity between ε_0 and the M4 preprocessor), Werner Koch, Marcus Brinkmann, Neal Walfield (who had graciously offered to proofread this thesis, even if in the end I finished too late to let him), Henrik Sandklef, Simon Josefsson, Juan-Pedro Bolívar Puente, Juan Antonio Añel Cabanelas, Reuben Thomas, Ralf the Autotools guy, Andreas Enge, Daiki Ueno, Neil Jerram, Stepan Kasal, Paolo Carlini. Christian Grothoff, Nathan Evans and the Lilypond guy showed genuine interest in epsilon and kept asking me about it in The Hague. And of course I can't forget Fred and George, who deserve to be mentioned for their crucial non-programming contribution to GNU recutils. When I was hit by a small emergency René Mérou was kind enough to lend me money after knowing me for less than one day, an act of trust justified by our common ideas. I still find this sense of commonality very moving.

Karl Berry was extremely patient in helping me with legal counseling about the epsilon copyright status; for this I also have to thank Christophe Fouqueré (again), Donald Robertson, the FSF lawyers, José Marchesi and Richard Stallman.

Thanks to the people at the FSF, FSFE, FFII, April and La Quadrature du Net. Keep going, friends. Even if we've not won all battles yet, we're writing history.

I'll be forever grateful to my masters, who changed my life. I've recently redis-

covered one of my first ones, Nemo Galletti, from whose work I learned procedural abstraction at the age of ten[1]. Some outstanding professors at the University of Pisa, particularly Marco Vanneschi and Giorgio Levi, shaped much of the good part of what I am now. From Marco Bellia, my Master's advisor in Pisa, I regret having learned less than I could have. I was inspired by the frighteningly talented OCaml people Xavier Leroy, Damien Doligez, Jérôme Vouillon, Pierre Weis; and most of all by the great masters I've never met: Abelson and Sussman, Chuck Moore, Paul Graham, Richard Gabriel. And I'll remember John McCarthy.
A long time ago when I was young and inexperienced, after reading what some people I admired were writing about Lisp, I decided to study it. I started with Steele's *CLtL2* [84], but without getting much at my first reading. I was to become a convinced Lisper only years later, after first understanding functional programming from *Functional Programming using Caml Light* [50] by Michel Mauny; that's the book from which I learned "everything". I was very impressed following my first meeting with Mr. Mauny in Paris, many years later, casually discussing with him without knowing his identity after somebody's seminar. Jean-Vincent asked me if I knew who that man who'd just left was. I said no. "Michel Mauny". I turned my head, but he was already gone. Now that this ordeal is over and I have no more conflicts of interest, I feel I can finally let Mr. Mauny know this. Thanks.

Among the other jury people I've particularly enjoyed the voluminous, useful and friendly feedback I got from Basile Starynkevitch — an unofficial third review. Along with Manuel Serrano, Basile actually tried the software and reported a couple minor bugs. Basile, one of these days I'll actually e-mail your own great master, as you suggested so many times.

And thanks to Emmanuel Chailloux, who even accepted to meet me *on August 17th* to have his paper copy of this document.

Juanma Díaz acquired and set up the server machine we share with Matteo and did the initial installation of the virtual system images running on top of it; later he also migrated the host a couple times. As a command-line guy I administer my virtual machine myself, but I have to recognize that it's also thanks to Juanma if my personal host `ageinghacker.net` has been more reliable than many "professional" servers.

Thanks to my landlords Tina and Mohamed Serhane for the rare virtue of being cheery and discreet at the same time. To Nadège. To Filippo Bellissima, still my favorite philosopher, for his good will. To James Randi (who has something interesting to say about PhDs) and the JREF people, for keeping up their mission against woo-woo. To Randall Munroe, Mohammed Jones and Scott Adams for their webcomics. Scott Pakin taught me a neat TEX hack on `comp.text.tex`, even if I

[1] The story at `http://ageinghacker.net/blog/posts/5` has a nice second part that I've not found the time to write yet.

didn't use it in this final version of the document.

Thanks to Richard Stallman for GNU (again) and particularly for GNU Emacs, to Donald Knuth for TEX, to Leslie Lamport for LATEX, to Lars Magne Ingebrigtsen for Gnus (using such a great client for e-mail and Usenet has actually made me a happier man), to Michael Stapelberg for `i3`; plus the thousands of other people who contributed.

Thanks to Wikimedia Foundation, to the fellow contributors to the English Wikipedia and to the contributors (I don't dare myself yet) to the French Wiktionary.

Thanks to the people at INRIA Saclay where I'm a postdoc now: Fabrice Le Fessant, who trusted me, and the colleagues Çağdaş Bozman, Pierre Chambart and Michael Laporte.

I apologize once more to Christophe Cérin, Jean-Paul Smets and Camille Coti for backing down from their postdoc offer at the last minute. If I hadn't had this unforeseen exciting opportunity at INRIA, I'd have been happy to work with them instead.

There are a few more people I'd really want to name, but since I'm honestly not sure if they would appreciate it I'll avoid mentioning them. In any case they will not mistake my respect of their privacy for an omission out of spite. There are indeed many people I'd be supposed to thank here "by custom" and yet are glaringly absent from this list *because they don't deserve to be in it* — which sums up pretty well what I think of customs; but I wouldn't imagine for one moment that people who are dear to me could doubt my sincere affection for them.

I regard the design of ε as a quite personal issue, which I think will be obvious from the document. It's the expression of my philosophy of what a programming language should be, for in the end the place in the design space where one chooses to look ends up being more a matter of personal preference than anything else. I've developed my opinions in years of reading and discussing, mostly influenced by the Scheme, Forth and Common Lisp communities. Interestingly, very few of the people working physically close to me ever seemed to share my views. When comparing opinions — which has been useful in any case — I've resisted interferences and rejected many suggestions; most strongly, the mistaken idea that static typing should always come at the expense of everything else. The final shape of the ε_0 and ε_1 languages as described here is a product of *my* ideas, something I'm willing to take responsibility for, pros and cons and all.

Jean-Vincent and Christophe, thanks for the freedom you gave me to explore and to do what I believed. All in all, it was fun.

Luca Saiu, December 2012

Contents

Chapter 1

Introduction

Reductionism is a viable strategy for designing and implementing practical programming languages, leading to solutions which are easier to extend, experiment with and formally analyze.

Contents

Programming languages have proliferated nearly since the beginning of Computer Science. However, despite the sheer number of dialects with different syntaxes and details, there is still comparatively little variety in programming models and paradigms — yet programming problems remain at least as hard as ever.

In order to really innovate in this field researchers need extensible languages which are easy to modify and experiment with, but at the same time not limited to simplified idealizations. Bringing the same idea out of the lab and into practice, an expert end-user should be able to bend and adapt the language to make it fit her problem, rather than the opposite.

For this to be possible a language has to start out simple and open-ended: able to express different paradigms, yet not hardwired for any; easy to reason about in a rigorous way when needed, without being unconditionally constrained.

1.1 Programming language taxonomy

Languages may be classified along at least three mostly orthogonal axes: *paradigm*, *typing policy* and *concurrency model*. In the following we limit ourselves to a quick overview; reviews articles such as [92] contain a much more detailed topology, with extensive examples.
Furthermore, not all of the relevant concepts have satisfactory formal definitions; but in this whirlwind tour we are going to renounce most pretenses of being exact, accepting to speak of very general concepts in terms somewhat vague.

1.1.1 Paradigm

Many popular languages such as C are *imperative*. Imperative languages, based on destructive mutation of state and explicit control flow, trace their origin to Turing machines, and in practice are easy to understand in terms of the underlying machine language.

Functional languages such as Haskell, ML and Lisp, shunning or at least limiting the occurrences of assignment statements, are basically sugared versions of some λ-calculus variant; their level of abstraction is much farther away from the hardware than imperative languages. Most functional languages are *higher-order*, i.e. they allow to pass functions as parameters to other functions, and to return functions as results.

At an even higher level, *relational* and *constraint* programming attempt to support a declarative, rather than algorithmic, style by dealing with sequences of data in an extensional fashion and having the user exploit data relations instead of building explicit data structures. Such languages tend to be based on particularly clean and simple mathematical theories such as relational algebra (the SQL query language) or some subset of the Predicate Calculus (Prolog[1]).

Object-oriented languages such as Smalltalk are more pragmatic: they encourage modelling data structures upon real-world entities by making the "behavior" or a computational object a function of the object identity, and making it easy to define related classes of objects by only specifying their differences.

Other families including *concatenative languages* such as Forth and Postscript, and *array languages* like APL can be more or less directly traced back to one of the four main groups.

1.1.2 Typing policy

Another orthogonal attribute of programming languages is their support for typing: programs written in *statically-typed* languages are mechanically analyzed prior to execution in order to check that some soundness property is satisfied, thus preventing certain errors from ever occurring at runtime: the compiler will simply reject any "suspicious" program — invariably including some false positives. ML and Haskell are examples of statically typed languages with *strong* type systems; many popular languages such as C, C++ and Java are also statically-typed, but their very complex semantics do not allow the extensive static checks which are relatively easy to

[1]Despite not being meant as general-purpose languages, we argue that database query languages are actually much better examples of declarative non-algorithmic programming than logic languages: query languages allow to reason about objects and their relations, *completely abstracting away* from data structures and even more importantly search strategies, i.e. algorithms. By contrast programming in Prolog in practice requires to constantly keep in mind its operational semantics, for reasons of efficiency and even correctness: for example just reordering two Horn clauses, which from a logic point of view simply yields an uninteresting equivalent variation, can easily change complexity from linear to exponential or dramatically alter termination properties and the number of results.

perform in functional languages[2], and many more runtime errors remain possible: we speak of *weak* static type checking.

By contrast *dynamically-typed* languages such as Lisp, Perl and Python perform checks *at runtime* before executing each operation subject to failure, typically at some cost in performance but gaining expressivity in comparison to the static-typing case; of course dynamic typing by itself cannot statically guarantee any soundness property.

Some low-level languages such as Forth and most assemblies are *untyped*: each datum is interpreted as-is without any check or conversion, assuming that it is always a valid operand for its operator; such languages trading safety for efficiency may not be suitable for all applications, yet they also definitely have a place in programming.

A subset of statically and strongly-typed languages including ML and Haskell employs *type inference* to automatically reconstruct type declarations from programs, rather than have the programmer provide them; this is convenient, but since type inference is undecidable for the most powerful type systems [18], relying on it alone reduces the expressivity of the language. Type inference is harder to employ in non-functional languages, and possibly because of cultural biases it is not widely used with weak type systems.

Even if mainstream languages seem resistant to adopt any such technique, the idea of static checking can be extended from typing to other properties computable via (necessarily partial for nontrivial languages) static analyses such as termination, time and space complexity, or escaping.

1.1.3 Concurrency model

Another aspect which we must at least cite constitutes another whole axis in the language space topology: the *model of concurrency* (synchronous versus asynchronous, message-passing versus shared state) also has a deep impact on the language semantics, and not only on the implementation of the runtime system.

The concurrency model of all the mainstream languages named above is *asynchronous shared-state*: concurrent "threads" read and mutate the same global state, explicitly synchronizing accesses when needed. The other important concurrency model is *message-passing*: threads or processes don't share state, but cooperate by exchanging messages. Erlang supports message-passing only; in the other languages mentioned above message-passing can be implemented on top of shared state, or is available as a thin "wrapper" over the inter-process communication primitives provided by the operating system. Languages with a *synchronous* concurrency level (at the software level) are a research topic [9, 17, 67] but have yet to see major application.

[2]It could be argued that the idea of passing parameters to and receiving results from a function lends itself to reasoning about compatibility; sequential side effects, on the other hand, always "compose well" with one another in a superficial sense, but may lead to more subtle violations of implied invariants.

Synchronous models are better suited to formal reasoning: in his formal calculi for concurrency CCS [56] and π-calculus [53, 54] Milner considered synchronous communication as primitive, and represented asynchronous processes by adding (synchronous) "queue processes".

On the other hand modern parallel hardware is strongly asynchronous, and truly parallel synchronous implementations on top of it tend to be prohibitively inefficient.

Older-generation languages tend to support no notion of concurrency at all.

1.2 Hybridization and complexity

Why are there so many programming languages? Couldn't they just agree on one? We have all heard this naïve question.

The fact that such a question can only come from a beginner is evident from our experience of how coding in even surprisingly similar languages "feels" different: for example about[3] the only real semantic difference between Pascal and C is the different *strength* of their type systems — static in both cases; yet the subjective "experiences" of writing in the two languages are *far* apart, as any programmer having used both can witness. That said, we also have to recognize that many differences between languages are in fact incidental, due to backward compatibility concerns or cultural inertia.

Another, deeper, answer to the beginner question is that different problems call for different languages. But then, why not merging the greatest possible number of features from different styles into one "perfect" language? In fact there exists such a trend: languages inspire and influence one another, and some recent ones such as Oz [93] make a point of offering support for *as many different paradigms as possible*; but even without looking at such extreme examples, a move towards *hybridization* is evident in most recent languages: contemporary languages such as C++, Java and C#, but also the popular "scripting languages" Python, Perl, JavaScript, incorporate at least *two* paradigms — imperative and object-oriented, with some elements of functional programming being more slowly accepted into the mainstream; in fact it could be argued that object-oriented programming is itself firmly rooted in the imperative paradigm, with only some restricted patterns taken from functional languages[4]. Most object-oriented languages also have hybrid type

[3] The different *funarg* problem [61] workarounds bear a much more limited impact in practice.

[4] The idea of *late binding* at the heart of object-orientation would be easy to emulate with data structures containing functions; in fact virtual method tables are typically implemented as chained arrays of pointers to functions, where functions have access to a "struct" holding field values: in other words, chained *closure* arrays: if method lookup fails in one class, a link is followed and the next one is tried, up the inheritance chain — or occasionally *sideways*, in case of multiple inheritance. As a different kind of hybridization, other non-imperative languages now include object-oriented features: the ML dialect OCaml [71, 19] managed to also add objects in a mostly-functional language and still keep its type system strong and static, at the cost of some complexity. More pragmatically, most modern SQL systems include some kind of object-oriented extension, more or less well-integrated in the relational paradigm.

systems, with some static *and* some dynamic checks. Often *both* message-passing and shared-state are available as concurrency models.

1.2.1 Hybridization limits

There are clear limits to hybridization: some features regarded as desirable in different communities are mutually incompatible, if not opposite. For example both having a static typing discipline and *not* having one may be reasonably argued to be useful features: one solution permits to prove run-time properties of a program before running it, the other improves expressivity. In the same spirit, the useful properties of purely functional languages [10, 40] would be destroyed by adding an assignment operator.

But even if we forget for a moment that many possible sets of features are just incompatible, designing a strongly hybrid language entails giving up on finding simple answers to programming problems, and just hoping that programmers will be better prepared for the unknown with a bigger toolbox: the bigger, the better. This pragmatic approach hits its limit when a language becomes too big to be *intellectually manageable* — at which stage the language may or may not be adequate for most tasks.

As a further objection against hybridization, working with such large chimeras makes harder to experiment with language features by building prototypes — in fact it may not be by chance that most such experimentation has historically taken place in Lisp dialects, which we will see to be the closest to our model.

1.3 *Growing a language*

Guy L. Steele dealt with the issue of the "size" of programming languages in his famous OOPSLA 1998 keynote talk "Growing a language" [85]. In a wonderfully deconstructionist exploit, Steele constrained his own English to follow the same rigid rules of formal languages in which *every non-primitive "word" needs to be explicitly defined before use*. By taking as primitives only English monosyllables he tried to communicate the feel of using a very small (programming) language.

The main point of the speech was the idea of working with a language powerful enough to be *evolved* by the user community under the coordination of a maintainer; and possibly even more important, *the user herself* would bend the language to her needs, as part of the daily practice of programming.

Even after several polysyllable definitions Steele's original prose retains its peculiar charm:

> *[...] a language design of the old school is a pattern for programs. But now we need to 'go meta.' We should now think of a language design as a pattern for language designs, a tool for making more tools of the same*

> *kind. [...] My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language.*
>
> — Guy L. Steele Jr., [85]

As the initial iteration of this process Steele proposed Java with some minor changes — thus at least a middle-sized language; in his opinion intentionally small languages such as the Lisp dialect Scheme, originally his own brainchild [89, 87], would remain hopelessly inadequate for modern tasks, as he tried to suggest with the one-syllable metaphor.

More than a decade has since passed, and the envisaged extension of Java by the community has not materialized[5].

The idea of "growing a language" remains a valid strategy, if not even the only realistic one. Without overlooking this important engineering insight, we find it worth to spend some words on what we do *not* agree with in Steele's presentation. Anyway, since much of the controversy will center around Java, to Steele's credit we must at least cite his later contributions based on Fortress [86], sharing the same idea of a "growth plan" but with a more suitable core language. That is the point: what makes Fortress a better match than Java for the task? And what yet superior alternative can we extrapolate from this trend?

1.3.1 Procedural and syntactic abstraction

In our opinion it is not by chance that the crucial insight for finding the missing ingredient was provided in [2], co-authored by Gerald J. Sussman — the other father of Scheme.

Our sketch of a language topology is reasonable but fails to really capture the actual expressive power of a language, as until this moment we have ignored a fundamental and orthogonal class of language features: *means of abstraction*, called "patterns" in Steele's quotation above.
Starting from the very first chapter, [2] speaks about means of abstraction as ways of *naming* patterns of code, possibly with parameters, so that they may be re-used at will as if they were primitive. In other words a mean of abstraction allows to *factor* away some code, so that we can reason at a higher level and ignore irrelevant details unless needed. The idea is by necessity vague as it potentially extends to any point of our language topology and beyond, yet we do not feel much danger of ambiguity: any programmer will promptly recognize abstraction features. We are speaking about *procedural abstractions* (including all the obvious generalizations to functions, predicates, modules and classes); and, as a separate group, *syntactic*

[5] Ironically, something close to Steele's vision of mostly-decentralized extensions has materialized *in Scheme* with SRFIs [80]; development is active, at least in the comparatively small scale of the Scheme community.

abstractions such as macros. No other example of a syntactic abstraction feature comes to mind and in fact no other case is in common use[6] — but we are going to introduce another kind in §1.3.3, and more fully in §5. Modern high-level languages support more or less adequate procedural abstractions, including higher order (C++ supports "lambdas" as per its latest Standard [38] and even Java should follow suit), but most are still very weak in syntactic abstraction; we are now proceeding to explain why this is important.

1.3.2 Syntactic abstraction and core-based languages: macros

In order to clarify what we mean by syntactic abstraction, we are now going to informally present a classic example.

Let us assume an imperative language similar to Pascal or C, with a `while`..`do` ..`done` loop but no `repeat`..`until`; we want to extend the language so that we can write:

```
1  procedure print_at_least_once (n):
2    variable x = 1;
3    repeat
4      print("x is ");
5      print(x);
6      newline();
7      x := x + 1;
8    until x > n;
9  end.
```

With a suitable macro system we could define `repeat`..`until` as syntactic sugar, so that for any sequence of statements s and any expression e, the loop "`repeat` s `until` e" is rewritten into "s`; while not (`e`) do` s`; done`". Handwaving away some trivial details which are not relevant here, we could say that the macro definition is:

```
1  macro repeat <s> until <e>:
2    <s>;
3    while not (<e>) do
4      <s>;
5    done;
6  end.
```

Using the `repeat`..`until` macro, the macroexpander stage of the compiler would automatically rewrite the above definition of `print_at_least_once` into:

[6] Some historical Lisp dialects used *fexprs* [64] as an alternative to macros; so does a new dialect called Kernel [78], resurrecting them thirty years later. Fexprs are also discussed at some length as an implementation device in [87, pp. 25-26].

```
1  procedure print_at_least_once (n):
2    variable x = 1;
3    print("x is ");
4    print(x);
5    newline();
6    x := x + 1;
7    while not (x > n) do
8      print("x is ");
9      print(x);
10     newline();
11     x := x + 1;
12   done;
13 end.
```

Unsurprisingly enough, the rewritten code contains a repetition: the macro call "factors away" an undesired regularity which would make the code harder to maintain if written directly in the extended form; even in such a trivial example as this the code using the macro looks easier to read, and its purpose more explicit. Also notice how macroexpansion had only *local* effect: the macro call has been replaced, but not its surrounding code.

It is worth stressing how our `repeat..until` loop functionality can *not* be defined as a procedure, unless the language supports higher order or some more exotic language feature such as passing statements as parameters; and even where those features were available, notice how macroexpansion might still produce a more efficient result and possibly be safer, as it takes place entirely *before* runtime.
Our sample macro simply glues together pieces of code without performing any substantial computation. This is not always the case: several languages, mostly Lisp dialects, have *Turing-complete* macro systems [79, 4, 47]. Other languages such as C are limited to weak token-based preprocessors [37] whose power does not reach much further than defining `repeat..until` (with uglier syntax). Other languages, Java included, do not support syntactic abstraction at all. We argue that precisely this weakness of Java has prevented Steele's plan from materializing.
Indeed the *bottom-up* programming style in which the language is extended to suit the problem, as implicit in Steele's quotation, is very typical in the Lisp world — and quite alien to most other communities.

Macros are so helpful in building syntactic sugar for existing languages that some Lisp dialects such as Scheme are in fact *core-based* [79, §1.9, §B]: implementations may choose to develop at a low level some core forms and define the rest of the language as a set of macros, eventually rewriting programs into combinations of core forms only.

As an obvious such example, a block in a higher-order functional language can always be rewritten into the immediate application of an anonymous function with the bound variables as its formals and the block body as its body, and with the bound expressions as actual parameters: "`let a = 1 + 2, b = 3 + 4 in a + b`"

has the same operational[7] behavior as "`(λ a b . a + b) (1 + 2) (3 + 4)`"; since the language needs λ and function application anyway, we can define `let` as a macro rather than having it as a primitive form, obtaining a simpler core language.
As the number of bound variables is arbitrary, the sample macro language we showed above can not express the rewrite quite adequately; yet the task is considered very ordinary in Lisp dialects, which to ease metaprogramming *use the same syntax for programs and data structures*. Deferring the explanation of details to §5, we show here a possible definition of `let` just to highlight its simplicity[8]:

```
(define-macro (let bindings . body)
  `((lambda ,(map car bindings)
      ,@body)
    ,@(map cadr bindings)))
```

This version of `let` binds variables "in parallel": defined expressions have no visibility of bound variables. The alternative "sequential-binding" block known as `let*` in Lisp is also easy to obtain by macroexpansion, in this case by rewriting it into nested trivial parallel-binding blocks in which the *outermost* `let` binds the *first* variable bound by `let*`. Recursive macros can still be very simple:

```
(define-macro (let* bindings . body)
  (if (null? bindings)
      `(let () ,@body)
      `(let ((,(caar bindings) ,(cadar bindings)))
         (let* ,(cdr bindings)
           ,@body))))
```

`let` follows the intended evaluation order under a *call-by-value* strategy: fully reducing all the operands before the application forces to evaluate the body only after all bound expressions. Much in the same way, `let*` constrains the evaluation order so that bound expressions are reduced top-to-bottom.

There are many other similar examples: the short-circuiting left-to-right versions of the `and` and `or` operators are easy to express with conditionals: "`a and b`" has the same behavior as "`if a then b else false`", while "`a or b`" can be rewritten into "`if a then true else b`".

[7] Despite the vagueness entailed by not having specified any particular semantics we have to at least recognize the fact that what is equivalent *operationally* might not be under a corresponding *static* semantics; in particular we might have good reasons for using different type rules in the case of the expanded form. Such an observation is not new at all: Milner had already recognized in [55, §3.5] what is now called *let-polymorphism* [63, §22.7]. Here we intentionally disregard any static semantics, delaying the justification to §4. This same remark also applies to the following examples in this subsection.

[8] We have omitted the check verifying that bindings are two elements long. Apart from that, the definition is perfectly realistic.

We can go even further: why having a conditional at all? By Church-encoding booleans so that "`true`" is "$\lambda xy.x$" and "`false`" is "$\lambda xy.y$", and taking "$\bullet$" as the element of a unit type (or indeed as any object), we can define "`if a then b else c`" as syntactic sugar for "`((a (`λz`.b) (`λz`.c)) `$\bullet$`)`", for some variable z not occurring free in `b` and `c`. Again, λ and the function application with $\bullet$ allow the conditional to reduce as expected also under call-by-value.

1.3.3 Transforms as syntactic abstraction

The *continuation* of a subexpression of a program is a function or procedure with side effects which, given the result of the subexpression, returns the final result of the program [89, 5].

Programs can be *automatically* rewritten into *Continuation-Passing Style*, a normal form making all continuations explicit as λ-terms. For example, let the continuation of $a + 2$ be κ; then by using one of the transformations in [46] we obtain $(\lambda x.(\lambda y.\kappa(x + y))2)a$ as its CPS version. It is not too difficult to see how both versions yield the same result:

- First we evaluate a, passing it to its continuation $\lambda x.((\lambda y.\kappa(x + y))2)$;
- the continuation of a binds it to x, and passes 2 to its continuation $\lambda y.(\kappa(x + y))$;
- the continuation of 2 binds it to y, evaluates the sum of x and y, and passes the result on to κ;
- κ provides the sum of x and y to the rest of the computation, which will finally yield the result.

One desirable feature of the CPS form is its independence from the evaluation strategy: in particular, the reduction sequence above holds for both call-by-value and call-by-name. Because of this and other useful properties, CPS may be convenient to use in compilers as an intermediate form to perform some semantic-preserving optimizations [83, 5, 44]. However, since our main interest here is program expressivity, we are particularly interested in *first-class continuations*: a syntactic form such as `call/cc` permits to access continuations *as data* in an untransformed program, performing "jumps" into or out from expressions [79]. The `call/cc` form can also be rewritten into an ordinary λ-term along with the rest, by the same transformation which turns the program into CPS.

First-class continuations are famously counter-intuitive and difficult to employ directly, but they can simulate powerful control features such as *exceptions*, *coroutines*, *generators*, and even *backtracking*; by using macros we can *syntactically abstract* away the implementation of these control features, and provide a simple high-level syntax.

A prerequisite for doing this is, of course, hiding the transformed program from the user. As already evident from the trivial example above, CPS-transformed programs are long and tedious to read: the user should simply use `call/cc` (or, better, syntactic forms reducing to `call/cc` uses) in *direct-style*, untransformed programs. We already stated that the transformation can be automatic, and that it supports `call/cc` as well — hence such a kind of syntactic abstraction is clearly possible. But *can we define a CPS transformation using macros?*
The answer is **no**: the result of CPS-transforming an expression depends on its *context*, while macros only have access to their parameters. If we are to support global program rewritings such as CPS, we need to introduce a second syntactic abstraction feature: we call it the *transform*[9]. Transforms map syntactic objects into other syntactic objects, and can be employed either before defining a global object, or retroactively on an existing program.

Transforms can be quite useful [46]: just as it is the case for CPS, the *Closure Conversion* process rewriting λ-terms into explicit *closures* [72, 6] may require contextual information: unless we want to close over globals[10], when building a closure we need to know the set of variables which are bound at that program point.

To reiterate, by composing two transforms it is possible *to build a language supporting first-class continuations based on a core not even containing anonymous λ-terms.*

Program transformations are commonly seen in formal mathematical presentations, but to our knowledge they have not been available as an abstraction tool in any general-purpose programming language up to this point.

1.3.4 Why reductionism

It should be clear by now that taking the core-based approach to its logical extreme yields a very simple core language. Anyway before committing to that route it may be worth to pause and consider our ultimate purpose, and of course the tradeoffs involved.

First of all a small core language is easy to reason about, particularly if program analysis is automated — which had better be, at a time when programs as written by humans can be millions of lines long. Moreover the core language will tend to

[9] Some controversy remains among English purists about the use of the term "transform" versus "transformation"; according to some, a transform is *the result of* a transformation. We admit that even in Computer Science the use of the term "transform" for *the function operating the transformation* is not universal, but we still prefer the shorter form.

[10] ML dialects do in fact close over "global" variables as well, but we find that this choice complicates interactive programming, sometimes yielding "unexpected" results in case of variable redefinitions. As a matter of personal taste we tend to prefer the solution of Common Lisp and Scheme which close over nonlocals only, not including globals. In practice we suppose that the ML behavior is dictated by the need not to invalidate the results of previous type inferences whenever a global is redefined.

be easier to "get right", as a small number of features will reduce the chances of unforeseen bad interactions. A naïve implementation will also be quick to build.

On the other hand, as the core language will not be usable in practice without several layers of extensions, the problem of tracking source locations becomes relevant: a user will normally write in the extended language and expect the programming system to refer the extended program *as she wrote it* in terms of file names, line numbers and syntactic forms: for example error messages referring the final transformed program would prove very hard to understand[11]. Just a little more subtly, static analyses will often need to refer to the non-primitive forms of some higher-level intermediate languages, instead of the core[12]: hiding details is the whole point of abstractions. The obvious implementation will also be inefficient, as evident from the examples in §1.3.2 and §1.3.3; however, since the code will start small and manageable, it will be less hard to introduce optimizations where needed.

We have mentioned pros and cons; in fact we argue that *all* the objections above can be answered, but even this it not essential in our view: we believe that any or all of the problems above would still be offset by the crucial advantage of obtaining an *open-ended* language, able to grow in unanticipated directions.

In order to achieve this ultimate goal we need *both* a small core language, and strong syntactic abstractions.

The choice above is a conscious restriction on the region of the design space we are setting to explore. Other choices are certainly possible, and a couple have been tackled in the past, with interesting results.

1.3.5 Related languages

The most famous example of this design as well as a source of inspiration for this work, again, is Scheme [79]: anyway we have to remark how the core contains much more than a functional language[13], including complex features such as first-class continuations, which could *not* be re-implemented by using Scheme's syntactic abstraction alone. Despite its beauty Scheme's syntactic abstraction system itself is

[11] Actually, simple partial solutions to this problem have been known for a long time. For example the C language preprocessor generates a `#line` directive in the output mentioning the original file name and line whenever the source of the output changes; this is enough information for the compiler to map each element of the single stream of code it receives back to its original location. Anyway debugging C code using macros remains notoriously hard: one reason is the lack of a similar output-to-input mapping at the level of syntactic forms.

[12] Again, an example is `let` under Hindley-Milner type inference, which could be defined with a macro such as the one above, but would benefit from being typed with *let-polymorphism*, differently from generic function calls. As another example, a CPS transform can encode *first-class continuations* into anonymous functions — but in a static type system, continuations would need their own separate typing rules.

[13] Tom Lord's failed proposal for the new Scheme Standard — before his alleged expulsion from Working Group 1 — would have been philosophically closer to our vision, despite the very different core. Lord's core language "WG0 Scheme" would have used *fexprs* [64, 78] and reified environments. His own recount is at `http://lambda-the-ultimate.org/node/3861#comment-57967`.

also quite complex, based as it is on hygienic macros [42]. The macro system does not look easy to factor, as shown by the experience of `psyntax`: `psyntax` is a compiler translating Scheme with macros into pure Scheme, which is itself written in Scheme *with macros* and hence needs to be bootstrapped with a pre-compiled version. `Psyntax` is elegant but, by the author's admission [25, §18], not trivial.

As our second example of a reductionistic design, Forth [30, 29] is about as obvious: its basic mechanism is imperative state mutation involving a fixed set of global stacks, with no need for actual expressions: "`42`" is *an imperative statement pushing a number on a stack*, and the "`+`" *word* replaces the two topmost stack elements with their sum. All words, predefined or not, are zero-parameter zero-result procedures with imperative stack effects. Even control features such as loops are defined as stack operations, involving the stack normally used for procedure return addresses.

Defining a word involves temporarily switching to a "compile state", in which each encountered word is taken from the program input stream and appended to the current definition rather than being executed immediately; to implement this, all word definitions begin and end with the state-switching words "`:`" and "`;`". The words[14] "`(`" and "`)`", respectively opening and closing a comment, work in the same way. No syntactic structure exists at a scale larger than individual words.

Forth is an unusual language somewhat defying classification, and it is debatable whether abstraction features such as changing "state" even count as syntactic — it might be the case that very strong procedural abstraction may partially compensate for the absence of syntactic abstraction features, like higher order does in functional programming[15]. Syntactic of not, Forth abstraction features have proved to be effective and they do allow to abstract to a high level, despite the language being often used on the bare metal without even an operating system. Due to its simplicity, like Lisp, Forth has been independently re-implemented many times, by building some core primitive words in assembly and then writing the rest of the system in itself. The language is so small that *hardware* implementations exist [43, 34], and its key proponents such as Chuck Moore exhibit a cultural tendency to reject all conventional software as bloated and hopelessly overcomplicated [60].

Finally some classic object-oriented systems such as Smalltalk are fairly minimal-

[14] 2016 correction: I have since learned that "`)`" in its role shown above is not actually a word. In actuality the definition of "`(`" discards all the input text up to and including the first "`)`" character, in a way analogous to, for example, "`s"`" and "`"`"; this solution avoids the need for the comment closing character to be delimited with whitespace. The comparison with "`:`" and "`;`" remains valid, despite the slightly different nature of the comment solution.

[15] Our favorite example is `with-mutex` (for example as in [21, §Mutexes and Condition Variables]) in a shared-state concurrent language with exceptions: `with-mutex` executes some given statements in a critical section so that a mutex is acquired at entry and released at exit, *including when an uncaught exception causes a jump out of the critical section*. It is easy to define `with-mutex` as a macro, and where macros are not available it can still be simulated using a higher-order procedure, at some loss of elegance. Without either of these features, the user is forced to duplicate code.

istic as well, and provide relatively strong procedural abstraction. Their example suggests *reflection* as a further strategy to help build complex programs: in object-oriented systems the state of each computational entity in the running program is available to the program itself, which can query objects for their interfaces at run time. Such runtime type information also provides the foundation of dynamic method dispatching, but we find this style of late binding to be less interesting for our purposes, serving better as a practical modelling tool than as a foundation for a core language.

1.4 Our solution

What kind of language do we want? In such a vast design space there is no clear answer, and committing a decision appears dangerous. Even with no breakthrough in sight at this particular time, our topology of §1.1 might even get enriched by entirely new dimensions in the future.

Of course we *do* have opinions about what kind of language would be better to solve the software crisis, and we will not even try too hard to hide our personal preferences as the description unfolds; but opinions are not science. Lacking a silver bullet, the best course of action seems to leave our design open-ended and follow in software the lesson of RISC, eschewing any particular focus or specialization in exchange for wider applicability.

In order to achieve Steele's vision in [85], our language:

- will be built on a very small core, like Forth, but in a form more amenable to formal reasoning;
- will provide strong syntactic abstraction features, like Scheme does, plus transforms, in the interest of expressivity;
- will provide reflection, like object-oriented systems;
- will not depend on either static or dynamic type checks in the core; such systems can be added as extensions;
- is meant to be practical, and efficiently implementable.

We call our language "ε", following the convention of naming small variables in Mathematical Analysis. When written in the Latin alphabet as "epsilon", the initial "e" in its name should always be lowercase.

A *personality* is a language made of the ε core plus extensions, in analogy with research operating systems implementing several different APIs on top of the same microkernel [96]. Personalities may reach very far from the core, as our transform examples above show.

We anticipate the development of complex[16] and widely diverging personalities, viewing the emergence of incompatible dialects, so feared in some communities, rather as a sign of health.

It is unfortunate but very possible that in a setting where a strongly extensible language is adopted, a new separation of programmers and meta-programmers (as personality developers) will follow the existing divide between programmers and language implementors; we do not claim to be able to cover such a cultural gap, but we mean to substantially ease the work of the second group.

The Programming Languages discipline needs more experimentation and prototyping. Let people play, and the language will grow.

1.5 Summary

Programming languages have traditionally been diverse in paradigm, typing policy and concurrency model. The current trend of hybridization makes languages more expressive, but also much harder to reason about; moreover most current languages remain difficult to extend and lacking in syntactic abstraction features.

In keeping with the philosophy of Scheme but bringing it much further, we propose the new programming language ε as an example of an alternative style of language definition in which strong abstraction capabilities allow the end user to express the needed linguistic features as translations into an extremely simple core language which is easy to reason about. A *personality* is a library of language extensions, in fact defining a new language in ε itself.

We argue in favor of language experimentation, recognizing dialect proliferation as beneficial in the long term.

[16] The composition of extensions is a difficult problem, for which no general solution is apparent: see §5.5.

Chapter 2

The core language ε_0

In this chapter we are going to formally describe the core language ε_0 by giving a small-step operational semantics for it and stating under which conditions an implementation is bound to behave according to the semantics.

As the foundation of much of the rest of this work, the specification will be used in §3 for describing the language reflective features, in §4 to prove correct a static analysis, and in §5 for defining syntactic extension semantics.

Contents

2.1 Features and rationale

Our core language ε_0 must be easy to reason about and efficiently compilable, has to include reflective features providing access to the program itself, and allow for parallelism. On the other hand the language does *not* need to be especially friendly to human users, since programmers will normally access it by means of higher-level syntactic extensions.

Satisfying such a set of requirements yields an idiosyncratic language whose extreme simplicity risks being overlooked at a first glance, obscured by some slightly unusual design choices.

Before formally specifying ε_0's syntax and semantics, it is worth to clarify the rationale of some design decisions.

2.1.1 First order

ε_0 is a first-order call-by-value expression-based imperative language with mutually-recursive procedures accepting zero or more parameters and returning zero or more results, where procedures are globally defined in a flat namespace.

The language is *first-order*: no anonymous procedures exist, and procedures can not be passed as parameters or returned as results[1].

Variable references are trivial to resolve: a block form is provided for binding *local* variables, which take precedence over procedure *parameters*, which in their turn take precedence over *global variables*. Since variables bound in other procedures are never accessible *no other scoping rule is necessary.*

Expressions return values and are allowed to have side effects; no looping form exists, and since recursion is only permitted at the top level among global procedures no explicit fixpoint operator is needed. This sets apart ε_0 from most functional languages, as the language of ε_0 expressions not referring global procedures is *not* Turing-complete.

The language exhibits relatively low-level features, making it easy to write a simple compiler with a clear efficiency model for non-self-modifying programs; Control Flow Analysis is also trivial, since all callees are explicitly identified by name at call sites. No escape mechanism such as exceptions, `longjmp` or first-class continuations is provided at this level, so evaluation strictly follows an intuitive stack discipline; in fact ε_0 is *stack-implementable*, and after macroexpansion and transforms have run, the residual ε_0 program does not necessarily require garbage collection.

2.1.2 Reflection

The current set of procedures is part of the global state of ε_0, and procedure definitions are accessible to the program for both reading and writing; this allows a program to analyze and modify itself.

Compilation in ε_0 consists in examining the current global state in terms of data and procedures, and producing as output a low-level program which, when executed, will reproduce the current state, in a style reminiscent of some Smalltalk systems and the Emacs `unexec` hack [47, §Building Emacs].

An ε_0 compiler can be an ordinary set of ε_0 definitions, running on top of the interpreter; in this sense we can say that the compiler, if any, is part of the program being compiled, rather than an external tool; and in the same way the user is free to build other *meta*-level tools such as code analyzers, transformers or optimizers.

2.1.3 Handles

Since a program has to reason about itself, ε_0 needs some mechanism for unambiguously referring to *program points*, also distinguishing different occurrences of otherwise identical syntactic forms. For this reason each syntactic form contains a

[1] We are going to relax this restriction in the implementation for efficiency's sake; anyway, as shown in §5.4.1.2, it will be trivial to automatically transform any program using "procedure pointers" into an equivalent first-order program. Closures will be remarkably more involved to define (see §5.4.4.4).

unique identifier which we call *handle*, the only requirement being that each expression of a program have a different handle.
At the implementation level it is reasonable to think of handles as unboxed integers or pointers to unique objects, but the specific nature of handles as a data type is immaterial: in practice the only relevant feature of a handle is its identity.

Handles are contained in expressions at all nesting levels, so that subexpressions at any depth may be referred by global names.

It is easy to associate information to handles, typically using global tables.

2.1.4 Primitives

The language specification should be complemented with a set of "predefined" primitive operators and data types for such operators to act upon, integer arithmetics being the obvious example; other useful primitives include memory allocation and side effects, and input/output. Primitives may accept parameters, return results and affect the global state but, the rationale being analyzability, they may *not* alter program control; this prevents "jumping" operations of the kind of exceptions, `longjmp` [37] and `call/cc` [79] from being implemented *as primitives*.

In the following we will not assume any particular set of primitives, limiting ourselves to some reasonable constraints which the particular primitives have to satisfy.

We do not dwell further on the specification of primitives which are to be implemented at low level, in practice using C or assembly language; a formal semantics of such low-level definitions is outside the scope of the present work. When working with any particular ε_0 or ε program, we will always take the set of available primitives as fixed.

2.1.5 Bundles

We allow ε_0 procedures and expressions to *return any number of results*, including zero; such as decision being more a concession to efficiency [8, 16], than an attempt at restoring symmetry between input and output.

A *bundle* is an ordered sequence of values which may be the result of a computation. The only feature distinguishing a bundle from an ordinary n-uple or list is the fact that bundles are not treated as data structures and in particular are *expressible but not denotable*: an ε_0 variable can only refer to *one* object, even if an expression is allowed to return a bundle of any size; the rationale being, of course, that no bundle data structures need to be expensively allocated and destroyed at runtime: each separate bundle element will be simply assigned a stack cell or register, possibly not even consecutively: no single "value" represents the whole bundle.

In this sense bundles bear resemblance to the the Common Lisp and Scheme *multiple values* feature [4, 79], with the important difference that in ε_0 callers do *not* ignore all results except the first one by default.

In order to work on bundle components, ε_0's block form binds up to *as many variables as the dimension of the bundle* that its bound expression evaluates to; hence ε_0 blocks also serve the purpose of "destructuring" bundles, which in practice simply means locally naming their components.

For example if a "`quotient-remainder`" primitive returned both the quotient and the remainder of two naturals (and indeed many hardware architectures provide such a machine instruction) a block could compute the quotient and remainder of some parameters, name the results respectively x and y, and evaluate a body in an environment where such variables are visible. It is worth to stress that at runtime this naming does *not* entail any moving, copying or – worse – memory allocation.

It is useful in practice not to always name *all* the components of a bundle, in particular for using nested blocks to simulate a statement sequence where the results of the intermediate steps are irrelevant; more in general, often one wants to ignore the result of a subexpression.

Bundles do complicate somewhat expression composition and are a possible cause of errors, but the performance gain they offer seems hard to obtain automatically by compiler optimization only. Recursive procedures returning more than one result seem a quite compelling example.

Of course personality implementors aiming for very simple extensions are always free *not* to use bundles, or for that matter any other ε_0 feature.

2.1.6 Parallel features

The parallel features of ε_0 appear mundane compared to some of the points above, limited as they are to creating *futures* associated to asynchronous *threads*, and extracting the result of a given future when waiting for its computation to terminate.

The system lends itself to both shared memory and message-passing, depending on primitives. In complex personalities aiming at high efficiency on large parallel machines or clusters, it is reasonable to expect that both styles will be used at different levels.

Again, parallel features introduce some complications into ε_0 but are too "fundamental" to be left out and then meaningfully reintroduced as language extensions.

2.2 Syntax

We are now ready to formally specify the syntax of ε_0 expressions, and establish some terminology about subexpressions.

Let the *set of variables* $\mathbb{X}$, the *set of procedure names* $\mathbb{F}$, the *set of primitive names* $\mathbb{\Pi}$, the *set of handles* $\mathbb{H}$ and the *set of thread identifiers* $\mathbb{T}$ be any numerable sets. By convention we will use the following metavariables, possibly with decorations, to represent objects of the respective sets: $x \in \mathbb{X}$ for *variables*, $f \in \mathbb{F}$ for *procedure names*, $\pi \in \mathbb{\Pi}$ for *primitive names*, $h \in \mathbb{H}$ for *handles*, and $t \in \mathbb{T}$ for *thread identifiers*.

Since the actual nature of "values" is irrelevant for the purposes of this chapter but has some other deep ramifications, we postpone its discussion until §3.3.1; as of now we simply speak of a *set of values* $\mathbb{C}$, using the metavariable $c \in \mathbb{C}$ for representing its elements.

For our examples in this chapter it will suffice to just use natural numbers, writing "$\mathcal{N}(n)$" to represent $n \in \mathbb{N}$, booleans $b \in \{\texttt{\#t}, \texttt{\#f}\}$ written as "$\mathcal{B}(b)$" *pointers* or memory addresses a written as "$\mathcal{A}(a)$" and *thread identifiers* or *futures* t written as "$\mathcal{T}(t)$".

Definition 2.1 (ε_0 syntax) *We define an ε_0 expression according to the following grammar:*

$e ::=$

x_h
$\mid c_h$
$\mid [\texttt{let}\ x^*\ \texttt{be}\ e\ \texttt{in}\ e]_h$
$\mid [\texttt{call}\ f\ e^*]_h$
$\mid [\texttt{primitive}\ \pi\ e^*]_h$
$\mid [\texttt{if}\ e \in \{c^*\}\ \texttt{then}\ e\ \texttt{else}\ e]_h$
$\mid [\texttt{fork}\ f\ e^*]_h$
$\mid [\texttt{join}\ e]_h$
$\mid [\texttt{bundle}\ e^*]_h$

We call each separate production right-hand side an expression form *or* form.

We call the first two cases a variable *and a* literal constant*, respectively. A* `let` *block contains zero or more distinct* bound variables*, a* bound expression*, and a* body*. A procedure call* `call` *expression mentions a* procedure name *and zero or more* actual parameters*. Very similarly, a* primitive call *mentions a* primitive name*, and zero or more actual parameters. The* conditional form `if` *comprises a* discriminand *expression, zero or more* conditional cases*, and finally the* then branch *and* else branch *expressions. A* fork *expression has the same syntax as a procedure call, while a* join *expression simply contains one* future *expression. A* bundle *expression contains zero or more* bundle items*.*

Each expression and its subexpressions, at all levels, contain unique handles*.*

We define `E` *to be the set of all ε_0 expressions; we use the metavariable e, possibly with decorations, to represent its elements.* □

The grammar in Definition 2.1 should be hardly surprising at this point, except possibly for the shape of the conditional and fork expressions.

The conditional expression shape is actually another small concession to efficiency: operationally, the discriminand expression is evaluated and compared to all the given conditional cases: if the discriminand evaluates to one of the given constants then the conditional reduces to the then branch, otherwise it reduces to the else branch. In many cases this kind of expression, when nested, is easy to optimize into multi-way conditional branches implemented as jump tables or balanced comparison trees.

Of course an ε_0 `if` expression can also simulate an ordinary two-way McCarthy conditional, by using a boolean literal as its only conditional case. Writing the *false* literal as `#f` in the style of Scheme, we can simulate the two-way conditional "$(e \to e', e'')$" by $[\texttt{if}\ e \in \{\mathcal{B}(\texttt{\#f})\}\ \texttt{then}\ e''\ \texttt{else}\ e']_{h_0}$, for some handle h_0. Of course reasonable personalities will define their own friendlier conditionals.

As for the `fork` expression, at a first look the form given above might appear gratuitously complicated compared to an alternative containing only one "asynchronous expression". Anyway such an alternative would be difficult to evaluate, as the asynchronous expression could then refer variables bound in the original thread, which would effectively become nonlocals. For this reason, as elsewhere in ε_0, we chose a more constrained syntax without too much fear of inconveniencing the user: personalities will provide higher-level fork operators.

Facilities for *defining* procedures will be dealt with in §3.

2.2.1 Meta-syntactic conventions for expressions

Since every syntactic object contains a handle, independently of the syntactic category or the specific case, when identifying particular components of a syntactic form instance we may explicitly specify a (meta-)handle in a *sub*-expression of a given expression, despite referring to the sub-expression itself just with a meta-variable; for example h_2 represents the handle of the `join` future expression in $[\texttt{join}\ e_{h_2}]_{h_1}$, regardless of the future expression specific "shape". We will also omit subscripts in meta-variables which already contain (meta-)handles unambiguously identifying instances: for example we will simply write $[\texttt{primitive} + e_{h_1}\ e_{h_2}]_{h_0}$ instead of the heavier $[\texttt{primitive} + e_{1h_1}\ e_{2h_2}]_{h_0}$. When only one identifier appears as a subscript, it should always be interpreted as a handle rather than a metavariable decoration.

We will usually name (meta-)handle indices in a top-to-bottom left-to-right order according to the expression syntax; we may let indices start from either 0 or 1, according to which option provides more notational convenience, such as avoiding the occasional "+ 1" in subscripts. For example we prefer writing an n-element multiple expression as $[\texttt{bundle}\ e_{h_1}...e_{h_n}]_{h_0}$ rather than as $[\texttt{bundle}\ e_{h_2}...e_{h_{n+1}}]_{h_1}$. Since we use handles only to identify occurrences of syntactic forms, their actual value is always immaterial: starting indices in meta-handle sequences can be just as arbitrary.

2.3 Semantics and the real world

Before finally specifying ε_0's semantics it is worth to add one last remark to prevent some misunderstandings, explicitly delimiting the cases in which an implementation is compelled to respect the semantics. This point is crucial if we are to speak about actual programs running in actual machines, rather than just another formal

calculus whose terms unfold into other terms in a Platonic universe where memory is infinite and checking for error conditions is for free.

As ε_0 is the underlying common language all personalities ultimately reduce to, it also represents *an efficiency upper bound*: a program written in a higher-level personality will only run as fast as its translation into ε_0; hence the critical need for speed, also offsetting the cost of making some implementations unfriendly and *unforgiving* of mistakes. But thankfully an unforgiving implementation does not need to be the only one, and when developing an application a user will benefit from the feedback of a *slower interpreter* failing in a more descriptive way than a "`Segmentation fault`" message, and possibly allowing some form of debugging.

The nature itself of failure needs to be carefully stated here: what we refer to as "failure" (§2.5.3) or "resource overflow" (§2.3.1) in the semantics does *not* necessarily translate into a dynamic check at the implementation level:

Implementation Note 2.2 (implementation guarantees) *A conforming implementation will behave according to the semantics provided that the execution never reaches an error configuration and never exceeds any resource limit; otherwise, the implementation behavior is unspecified.* □

Not failing and not overflowing resources is just a *sufficient* condition for an implementation to respect the semantics; in the implementation a program violating one of these condition is allowed to crash or silently return any result, possibly even the correct one: *no guarantees at all.* If a personality implementor wants to specify some behavior in one of these cases then it's her responsibility to perform *static checks on the input code* or to include *dynamic checks in the generated ε_0 code*, in order to prevent the conditions for unspecified behavior from occurring.

It may be worth to state explicitly that, as a trivial consequence of Implementation Note 2.2, an execution consuming an unbounded quantity of *any* resource has unspecified implementation behavior.

2.3.1 Resource limits

As it is easy to imagine, our first mention of *numerable sets* in §2.2 already hid a caveat: only a finite number of distinct values will be representable in an implementation, due to the finite nature of address spaces and word sizes.

There exist other remarkable cases of resource limits: for example the amount of available virtual memory, often dramatically smaller than what the address space allows for; operating systems also usually constrain the number of concurrent threads; an implementation might limit the stack size to a constant value. All of these cases will be covered by Implementation Note 2.7.

In the following we will state which resources may be limited by implementations in explicit *Implementation Notes* such as the following one; as already explained in

Implementation Note 2.2, an implementation is not forced to detect the failure at runtime, and may proceed with undefined behavior in case of resource overflow.

Implementation Note 2.3 (Syntactic resource limits) *Each instance of the following items occupies some memory in an implementation; an implementation will pose a limit to the sum total of all the used memory at any given time, possibly multiplied by a logarithmic factor.*

- *variable and procedure names, the cost being proportional to the name length;*
- *the number of handles;*
- *expression syntactic complexity.* □

In Implementation Notes dealing with resource limits such as Implementation Note 2.3 above, we deliberately ignore constant terms: for example if the physical resource occupation of n items of a certain kind is $n \cdot a + k$ units, an implementation may simply declare each item to take a units and the total resource availability to be k units lower than its actual dimension.
Moreover, since avoiding resource overflows is only a sufficient condition for an implementation to respect the stated semantics, we allow Implementation Notes to describe resource occupation *as an upper bound*.

At the cost of sounding pedantic, we stress that the statement above *does not* limit our reasoning about resources to asymptotic approximations; in fact, where a given implementation instantiates the precise costs and resource availability, it is possible to reason about whether a program "fits" the implementation on which it runs — the rationale of course being that a program overflowing resources is not any better than an incorrect one.

2.4 Configurations

We are now ready to formally define the mathematical structures used in ε_0's dynamic semantics.

2.4.1 The global state

A *global state* or simply *state*, always represented with a possibly decorated Γ metavariable, represents *the instantaneous condition of an execution*; an execution may access, reading and also destructively mutating parts of a state. We call "$\mathbb{\Gamma}$" the set of all possible states.

A state is a inherently composite object made of several *state environments*; by "environment" we simply mean a mathematical function mapping keys into values. We do not list all state environments here, since the need of some of them will not be apparent until later. In order to lighten our notation and to allow for yet-unspecified components without depending on some arbitrary order, we also avoid traditional projection and update operators, opting instead for a notation referring components *by name*, as if the state were a single "record" of environments.

2.4.1.1 Notational conventions for states and environments

It is often convenient to exploit the *set-of-pairs* nature of relations to represent environments in an extensional style, as in "$\{x_1 \mapsto q_1, ..., x_n \mapsto q_n\}$"; an interesting particular case is the empty environment, which is to say a nowhere-defined function, which we write as the empty set "$\varnothing$".

If ϑ is the state environment named n in Γ then we may write Γ_n to mean ϑ; and of course we also employ the ordinary notation for function application by writing, for example, $\Gamma_n(\mathtt{x}) = q$ or equivalently $\Gamma_n : \mathtt{x} \mapsto q$.

As per the standard update notation, we write "$\vartheta[\mathtt{x} \mapsto q]$" to represent an environment equal to ϑ everywhere on its domain except on $\mathtt{x}$, which the updated environment maps to q.

Extending the standard notation, we will also deal with *updated environments in the state*: in other words we build a state identical to a given one save for one environment, which has been updated in its turn; we will write "$\Gamma[^{\mathtt{x} \mapsto q}_{n}]$" to represent the updated state identical to Γ except for the environment named n, which will be $\Gamma_n[\mathtt{x} \mapsto q]$ instead of Γ_n.

We also write $\Gamma[^{\vartheta}_{n}]$ to represent a state identical to the state Γ except for the state environment named n, entirely replaced by the environment ϑ.

Our use of brackets for updated states is distinct from the usual environment update notation, which we *also* adopt: we write "$\eta[\xi]$" to mean an environment identical to η everywhere except on the domain of ξ, where instead it is identical to ξ.

Notice that, unless we are dealing with *meta*-labels such as "n" here in §2.4.1.1, we always write state environment labels in typewriter font: this makes it clear that we are establishing a label for some state environment at its first mention, without the need of detailing every time how one label represents the similarly-named state environment, when the association is always obvious from the context anyway.

2.4.2 Global and local environments

The *global environment* is a state environment mapping global variable names into values, and can be thought of as a partial function $\mathbb{X} \rightharpoonup \mathbb{C}$.

The global environment keeps track of globally-visible objects (*globals* or *non-procedures*), which are always accessible by a variable name unless shadowed by a procedure parameter or a local variable, which instead are bound in *local environments*: local environments, also $\mathbb{X} \rightharpoonup \mathbb{C}$ functions, take precedence over the global environment, and or course they are *not* state environments; we use the ρ metavariable for local environments, possibly with decorations.

For example, when evaluated in a state $\Gamma[^{x \mapsto \mathcal{N}(42)}_{\mathtt{global\text{-}environment}}]$ and the local environment $\varnothing$, the variable x in the expression x_{h_0} will refer the value $\mathcal{N}(42)$; but if instead the local environment was $\{x \mapsto \mathcal{N}(10)\}$, then $\mathcal{N}(10)$ would take precedence

over the global value in x.

2.4.3 Memory

ε_0 expressions are allowed to perform imperative operations on mutable data structures: in particular expressions may *read* or *update* cells of memory buffers, which can be *allocated* and *destroyed*.

Such operations rely on the `memory` state environment $\mathbb{A} \rightharpoonup \mathbb{C}^*$ as mapping *addresses* into mutable word sequences or *buffers*; we might occasionally refer to each buffer element as a *memory cell*.

It is important to notice that the memory state environment models *the heap* in the implementation, of which each cell makes up *one word*: here we are dealing with cells which can be allocated and destroyed with any strategy, rather than a simple LIFO policy.

No data structures such as conses, tuples and arrays are hardwired in ε_0, but memory makes it easy to define such objects in a personality. The fact that dynamically-created structures are "made of" memory entails their mutability in a natural way. Immutability, if one chooses to enforce it for some class of data in a high-level personality, can be realized with dynamic or static checks which prevent updating[2] — but in ε_0 all memory cells are freely mutable, so as not to restrict the user in any way.

At this point the reader may already be suspecting that the global environment could be used for simulating memory; while —assuming the availability of certain primitives— that intuition is correct, §3.2.2 will provide a strong argument in favor of having a separate memory state environment.

2.4.4 Procedures

A state also keeps track of the current set of procedure definitions.

The *procedure state environment* is a $\mathbb{F} \rightharpoonup (\mathbb{X}^* \times \mathbb{E})$ partial function mapping each procedure name into a pair holding its zero or more formal parameters and the procedure body; for example, if a procedure named f has formals $x_1 ... x_n$ and body e_h in the state Γ we write "$\Gamma_{\texttt{procedures}} : f \mapsto (\langle x_1 ... x_n \rangle, e_h)$"; we may also omit angle brackets when no ambiguity can arise, in this case writing "$\Gamma_{\texttt{procedures}} : f \mapsto (x_1 ... x_n, e_h)$".

We remind the reader that, since ε_0 is a first-order language and all procedures are global *no nonlocals can exist*; for this reason there is no need for closure environments at this level.

Up to this point we have dealt with the syntax of ε_0 expressions only, stating that

[2] A more radical strategy could involve a syntactic "extension" un-defining or otherwise making inaccessible the operators needed for the update.

global mutually-recursive procedures are also somehow available but without specifying any way of *defining* them. Because of interactions with the rest of the system the issue turns out to be more delicate than one could imagine, and we defer its full treatment to §3; what we can hint at now is that procedures can be defined with *primitives* and other procedures, as explained below.

2.4.5 Primitives

We call *primitives* a set of low-level routines accessible from ε_0 expressions, used for computation, program reflection or side effects. Primitives range from simple arithmetic operations such as `+` to reflection and procedure definition operations, potentially also involving destructive state updates.

In an implementation primitives are routines implemented in a low-level language such as C or directly in Assembly. This does not mean however that a primitive is allowed to "do anything": primitives must not disrupt the program control flow by performing jumps or non-local exits or reentries *à la* `longjmp` or `call/cc`: primitives may affect the global state but have to behave in a procedural fashion, always giving control back to their caller; primitive behavior can actually be modeled by *partial functions* taking a fixed number of parameters and returning a fixed number of results — including an input and output state. Such higher-order functional specification is a consequence of the fact that primitives, unlike procedures, are not directly implemented in ε_0 and hence lack high-level bodies or any treatable "source".

A *primitive function with in-dimension n and out-dimension m* ($n, m \in \mathbb{N}$ and $n, m \geqslant 0$) is a partial function $(\mathbb{C}^n \times \mathbb{\Gamma}) \rightharpoonup (\mathbb{C}^m \times \mathbb{\Gamma})$, mapping an n-uple of values and a state into an m-uple of values and another state; a *primitive* is a triple comprising the primitive function, its in-dimension and out-dimension, which respects Axiom 2.10. We call $\mathbb{P}$ the set of all primitives, with $\mathbb{P} \subset \bigcup_{n,m\in\mathbb{N}}\{\langle p, n, m\rangle \mid p \in (\mathbb{C}^n \times \mathbb{\Gamma}) \rightharpoonup (\mathbb{C}^m \times \mathbb{\Gamma})\}$.

The *primitive environment* state environment $\mathbb{\Pi} \rightharpoonup \mathbb{P}$ maps each primitive name into a primitive.

Axiom 2.10, defined in §2.5.3 and only needed for technical reasons, will just affirm that primitive success and failure are mutually exclusive.

From now on we will bend our notation a little further by writing "$\Gamma_{\texttt{primitives}}(\pi)(c_1, ..., c_n, \Gamma) = \langle c'_1, ..., c'_m, \Gamma'\rangle$" or "$\Gamma_{\texttt{primitives}}(\pi) :_{\#} n \to m$" as needed, to avoid useless pedantries such as "$p(\langle c_1, ..., c_n\rangle, \Gamma) = (\langle c'_1, ..., c'_m\rangle, \Gamma')$ where $\Gamma_{\texttt{primitives}}(\pi) = (p, n, m)$".

As an example, considering the `quotient-remainder` primitive of §2.1.5 in some state Γ we could write "$\Gamma_{\texttt{primitives}}(\texttt{quotient-remainder})(\mathcal{N}(13), \mathcal{N}(3), \Gamma) = \langle \mathcal{N}(4), \mathcal{N}(1), \Gamma\rangle$" meaning that the quotient and remainder of the naturals 13 and 3 are (respectively) the naturals 4 and 1, and that the primitive does not affect the global state; we could also write "$\Gamma_{\texttt{primitives}}(\texttt{quotient-remainder}) :_{\#} 2 \to 2$", by

which we would mean that `quotient-remainder` has two parameters and two results — which does not prevent the primitive function from being partial, as indeed it is. Where the particular state is obvious from the context or irrelevant, we even write "$\pi :_{\#} n \to m$" to mean $\Gamma_{\texttt{primitives}}(\pi) :_{\#} n \to m$, for the appropriate Γ.

As a further and possibly more interesting case, we just hint at the fact that *memory operations* such as allocating buffers and loading and storing words are performed by appropriate *primitives*: this will let us keep the semantics simple, ignoring the details of memory, and treating memory operations as just another instance of effects on the global state.

Specifying a complete set of "default" primitives is out of the scope of this work, but §5 will informally introduce most primitives currently used in the implementation, while §3 will deal with reflection and program-updating in relation with primitives.

We may informally speak of *applicables*, when abstracting away the distinction between procedures and primitives.

2.4.6 Holed expressions

In our dynamic semantics we need to capture intermediate computation snapshots in which an expression is in the middle of being evaluated.

We define below an extended ε_0 expression grammar, where the *hole* "$\Box$" stands for a subexpression which is yet to be fully evaluated.

Definition 2.4 ($\boldsymbol{\varepsilon_0^\Box}$ syntax) *We define the set $\mathbb{E}_\Box$ of possibly-holed expressions or "$\varepsilon_0^\Box$ expressions" by the following grammar:*

$e_\Box ::=$

e
$\mid [\texttt{let}\ x^*\ \texttt{be}\ \Box\ \texttt{in}\ e]_h$
$\mid [\texttt{call}\ f\ \Box]_h$
$\mid [\texttt{primitive}\ \pi\ \Box]_h$
$\mid [\texttt{if}\ \Box \in \{c^*\}\ \texttt{then}\ e\ \texttt{else}\ e]_h$
$\mid [\texttt{bundle}\ \Box]_h$
$\mid [\texttt{fork}\ f\ \Box]_h$
$\mid [\texttt{join}\ \Box]_h$

Syntactic cases are respectively named: non-holed expression, holed block or holed `let`*, holed call or holed procedure call, holed primitive or holed primitive call, holed conditional, holed* `bundle`*, holed* `fork` *and holed* `join`*.*

All cases save the first represent properly holed expressions. ▫

Notice that holes do not occur in all possible expression contexts: this issue is related to *tail contexts*.

Moreover, as the nonterminal $e_\Box$ never occurs in a production right-hand side, no holed expression can contain other properly holed expressions: this "single hole"

property reflects ε_0's deterministic sequential evaluation strategy.

2.4.7 Stacks

Rather than resorting to the traditional small-step semantics style [99, §2.6] in which the computed parts of an expression are replaced with values, here we adopt a more realistic and lower-level model using explicit stacks and keeping track of "return points"; this should already be clear at this point from §2.4.6.

We keep two separate aligned stacks per thread for describing evaluation, one stack representing the dynamic nesting of partially-evaluated expression forms and the other representing the dynamic nesting of values; we respectively call them the *main stack* or even simply *the stack*, and the *value stack*:

- The main stack is a sequence of pairs, each pair containing a holed expression and its associated local environment (§2.4.2): the set of all possible main stacks is $\mathbb{S} \triangleq (\mathbb{E}_{\Box} \times (\mathbb{X} \rightharpoonup \mathbb{C}))^*$;
- The value stack is a sequence of objects, each of which being one of a value, the *value separator* "$\wr$", or the *activation separator* "‡". Value stacks belong to $\mathbb{V} \triangleq (\mathbb{C} \uplus \{\wr, ‡\})^*$.

A two-stack solution is particularly appropriate because of bundles and is visually intuitive, but of course efficient implementations for conventional machines will reasonably use a single stack per thread.

We write stacks *horizontally, with the top on the left*: this is analogous to list syntax in Lisp and functional languages, and the opposite of Forth conventions.

We usually represent main stacks with the metavariables S and value stack with the metavariables V, possibly decorated.

2.4.8 Futures

As we have already hinted at in §2.4.7 and will be made more clear in §2.5, evaluation in ε_0 needs *two stacks per thread*, along with the global state.

The "main" thread of a computation is called the *foreground* thread; the global state holds information about *all the others*.

We call *future state environment* the state environment `futures` holding thread information. Such an environment simply maps each thread identifier into its stack and value stack, and belongs to $\mathbb{T} \rightharpoonup (\mathbb{S} \times \mathbb{V})$.

Implementation Note 2.5 (global state resource limits) *In an implementation the following resource limits hold:*

- *each global environment binding occupies a constant amount of the memory resource;*

- *each memory cell occupies a constant amount of the memory resource;*
- *each defined procedure occupies a constant amount of the memory resource;*
- *each defined primitive occupies a constant amount of the memory resource;*
- *each thread which is either running or being waited by a* `join` *expression in a background or foreground thread (§2.5.1) occupies a constant amount of memory.*

  An implementation may also limit the number of threads running or being waited for (as above) existing at any given moment, independently from memory usage.

In all the cases above, some implementations may also scale the total amount of occupied resource by a logarithmic factor. □

2.4.9 Configurations

A *configuration* contains information about the *foreground thread*, and a global state; of course the global state, among the rest, holds information about the other "*background*" threads.

The set of all configurations is $\mathbb{S} \times \mathbb{V} \times \mathbb{\Gamma}$; we usually represent configurations with the letter χ, possibly decorated; since configurations are potentially complex when we show their three components we always omit commas to reduce the visual clutter.

Evaluating a given expression e_h in a given state Γ entails building an *initial configuration* $(e_h, \varnothing) \wr \Gamma$: An initial configuration always has a main stack made by just a non-holed expression coupled with an empty local environment, and a value stack made of just one "$\wr$" separator.

Final success configurations contain an empty main stack and a value stack $\wr c_n c_{n-1} ... c_2 c_1 \wr$ holding the zero or more elements of the result bundle in a reversed sequence, preceded and followed by a "$\wr$" separator — the bundle inversion phenomenon being a consequence of the LIFO evaluation style.

For example, assuming a "reasonable" `+` primitive and some state Γ, we expect that by evaluating starting from the initial configuration $([\texttt{primitive +}\, \mathcal{N}(2)_{h_1}\ \mathcal{N}(3)_{h_2}]_{h_0}, \varnothing) \wr \Gamma$ we eventually reach a final success configuration $\langle\rangle \wr \mathcal{N}(5) \wr \Gamma'$; it is possible to have $\Gamma \neq \Gamma'$ because of background threads already started in Γ.

2.5 Small-step dynamic semantics

We are now finally ready to specify ε_0's dynamic semantics.

2.5.1 Small-step reduction

We need to formalize the intuitive notion of reduction. Given two configurations χ and χ', we say that χ *reduces to* χ' and we write "$\chi \longrightarrow_{\mathbb{E}} \chi'$" according to the following definition:

Definition 2.6 (small-step reduction) *We define the small-step evaluation relation*
$\_\_\_ \longrightarrow_{\mathbb{E}} \_\_\_ \subseteq (\mathbb{S} \times \mathbb{V} \times \mathbb{\Gamma}) \times (\mathbb{S} \times \mathbb{V} \times \mathbb{\Gamma})$ *according to the rules at pp. 32-33. In the rules we always assume* $n \geqslant 0$*, with the convention that an indexed sequence with left index 1 and right index 0 is empty.*

Each rule has associated a name, written on the left in brackets. □

$$[constant]\ \frac{}{(c_h,\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} S\ \wr c \wr V\ \Gamma}$$

$$[variable]\ \frac{}{(x_h,\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} S\ \wr c \wr V\ \Gamma}\ \Gamma_{\texttt{global-environment}}[\rho] : x \mapsto c$$

$$[\texttt{let}_e]\ \frac{}{([\texttt{let}\ x_1 ... x_n\ \texttt{be}\ e_{h_1}\ \texttt{in}\ e_{h_2}]_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho).([\texttt{let}\ x_1 ... x_n\ \texttt{be}\ \square\ \texttt{in}\ e_{h_2}]_{h_0},\ \rho).S\ \wr V\ \Gamma}$$

$$[\texttt{let}_c]\ \frac{}{([\texttt{let}\ x_1 ... x_n\ \texttt{be}\ \square\ \texttt{in}\ e_{h_2}]_{h_0},\ \rho).S\ \wr c_m c_{m-1} ... c_2 c_1 \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_2},\ \rho[x_1 \mapsto c_1, x_2 \mapsto c_2, ..., x_n \mapsto c_n]).S\ \wr V\ \Gamma}\ m \geqslant n$$

$$[\texttt{call}_e]\ \frac{}{([\texttt{call}\ f\ e_{h_1} ... e_{h_n}]_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho)...(e_{h_n},\ \rho).([\texttt{call}\ f\ \square]_{h_0},\ \varnothing).S\ \wr \ddagger V\ \Gamma}$$

$$[\texttt{call}_c]\ \frac{}{([\texttt{call}\ f\ \square]_{h_0},\ \rho).S\ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr \ddagger V\ \Gamma \longrightarrow_{\mathbb{E}} (e_h,\ \rho[x_1 \mapsto c_1, x_2 \mapsto c_2, ..., x_{n-1} \mapsto c_{n-1}, x_n \mapsto c_n]).S\ \wr V\ \Gamma}\ \Gamma_{\texttt{procedures}} : f \mapsto (x_1 ... x_n, e_h)$$

$$[\texttt{primitive}_e]\ \frac{}{([\texttt{primitive}\ \pi\ e_{h_1} ... e_{h_n}]_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho)...(e_{h_n},\ \rho).([\texttt{primitive}\ \pi\ \square]_{h_0},\ \varnothing).S\ \wr \ddagger V\ \Gamma}$$

$$[\texttt{primitive}_c]\ \frac{}{([\texttt{primitive}\ \pi\ \square]_{h_0},\ \rho).S\ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr \ddagger V\ \Gamma \longrightarrow_{\mathbb{E}} S\ \wr c'_m c'_{m-1} ... c'_2 c'_1 \wr V\ \Gamma'}\ \Gamma_{\texttt{primitives}}(\pi)(c_1, ..., c_n, \Gamma) = \langle c'_1, ..., c'_m, \Gamma' \rangle$$

$$[\texttt{if}_e]\ \frac{}{([\texttt{if}\ e_{h_1} \in \{c_1 ... c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho).([\texttt{if}\ \square \in \{c_1 ... c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h_0},\ \rho).S\ \wr V\ \Gamma}$$

$$[\texttt{if}_c^{\in}]\ \frac{}{([\texttt{if}\ \square \in \{c_1 ... c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h_0},\ \rho).S\ \wr c \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_2},\ \rho).S\ \wr V\ \Gamma}\ c \in \{c_1 ... c_n\}$$

$$[\texttt{if}_c^{\notin}]\ \frac{}{([\texttt{if}\ \square \in \{c_1 ... c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h_0},\ \rho).S\ \wr c \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_3},\ \rho).S\ \wr V\ \Gamma}\ c \notin \{c_1 ... c_n\}$$

$$[\texttt{bundle}_e]\ \frac{}{([\texttt{bundle}\ e_{h_1}...e_{h_n}]_{h_0},\ \rho).S\ \ \wr V\ \ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho)...(e_{h_n},\ \rho).([\texttt{bundle}\ \square]_{h_0},\ \varnothing).S\ \ \wr\ddagger V\ \ \Gamma}$$

$$[\texttt{bundle}_c]\ \frac{}{([\texttt{bundle}\ \square]_{h_0},\ \rho).S\ \ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr\ddagger V\ \ \Gamma \longrightarrow_{\mathbb{E}} S\ \ \wr c_n c_{n-1} ... c_2 c_1 \wr V\ \ \Gamma}$$

$$[\texttt{fork}_e]\ \frac{}{([\texttt{fork}\ f\ e_{h_1}...e_{h_n}]_{h_0},\ \rho).S\ \ \wr V\ \ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho)...(e_{h_n},\ \rho).([\texttt{fork}\ f\ \square]_{h_0},\ \varnothing).S\ \ \wr\ddagger V\ \ \Gamma}$$

$$[\texttt{fork}_c]\ \frac{}{([\texttt{fork}\ f\ \square]_{h_0},\ \rho).S\ \ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr\ddagger V\ \ \Gamma \longrightarrow_{\mathbb{E}} S\ \ \wr \mathcal{T}(t) \wr V\ \ \Gamma[^{t \mapsto ((e_h,\ \rho[x_0 \mapsto \mathcal{T}(t), x_1 \mapsto c_1, ..., x_n \mapsto c_n])\ \wr)}_{\texttt{futures}}])}\ \text{fresh } t,\ \Gamma_{\texttt{procedures}} : f \mapsto (x_0...x_n, e_h)$$

$$[\texttt{join}_e]\ \frac{}{([\texttt{join}\ e_{h_1}]_{h_0},\ \rho).S\ \ \wr V\ \ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho).([\texttt{join}\ \square]_{h_0},\ \rho).S\ \ \wr V\ \ \Gamma}$$

$$[\texttt{join}_c]\ \frac{}{([\texttt{join}\ \square]_{h_0},\ \rho).S\ \ \wr \mathcal{T}(t) \wr V\ \ \Gamma \longrightarrow_{\mathbb{E}} S\ \ \wr c_t \wr V\ \ \Gamma}\ \Gamma_{\texttt{futures}} : t \mapsto (\langle\rangle,\ \wr c_t \wr)$$

$$[\|]\ \frac{S_t\ \ V_t\ \ \Gamma \longrightarrow_{\mathbb{E}} S'_t\ \ V'_t\ \ \Gamma'}{S\ \ V\ \ \Gamma \longrightarrow_{\mathbb{E}} S\ \ V\ \ \Gamma'[^{t \mapsto (S'_t,\ V'_t)}_{\texttt{futures}}]}\ \Gamma_{\texttt{futures}} : t \mapsto (S_t,\ V_t)$$

It is easy to classify rules into four sets according to the holed expression case in the top stack pair, if any. We have:

- the *basic rules* [*constant*] and [*variable*];
- *expansive rules*, one per non-holed expression case, named after the form with an "e" subscript;
- *contractive rules*, one per holed expression case (except for the conditional, which needs two contractive rules), named after the form with a "c" subscript;
- the *parallel rule* [||] in the end, standing apart from all the others.

The core ideas of the evaluation are simple, and strongly rooted on the inductive nature of the expression syntax. Rule groups help to highlight the quite pleasant symmetry of the system:

- *base*: we evaluate a "basic" expression found on the top of the stack by popping it and pushing a corresponding value onto the value stack;
- *expansion*: before we can evaluate a non-basic expression on the top of the stack we need to evaluate its sub-expressions: so we replace the expression with its holed counterpart, and push its subexpressions on the stack on top of it, in an order such that the *first* one to be evaluated end up *on the top*;
- *contraction*: if a holed expression is on the top of the stack, this means that we have just finished evaluating its subexpressions: pop their values from the value stack, pop the holed expression from the stack, and proceed: according to the case this can mean pushing a further subexpression onto the stack or pushing results onto the value stack;
- *parallelism*: the parallel rule lets us concurrently perform a reduction in a background thread, whenever possible.

The *LIFO* policy outlined above enforces a rigid *call-by-value, depth-first left-to-right evaluation strategy*. We find that having such a simple and predictable evaluation order is very useful for both programming and reasoning about programs.

[*constant*] is trivial.

In [*variable*] it should be noted how the (topmost) local environment prevails over the current global environment in the variable rules. Of course the rule cannot fire if the variable is unbound.

[$\texttt{let}_e$] is simple enough: a `let` block is evaluated by first pushing the `let`-bound expression e_{h_1}; when such evaluation eventually ends producing a bundle in the value stack the `let` *contractive rule* can fire, assuming the bundle dimension is sufficient: then the holed `let` expression is replaced by the `let` body on the stack, with an updated environment in which the first n bundle components are named, and all m of them are popped off the value stack, implementing the behavior described in

§2.1.5. It should be remarked that the holed `let` expression "disappears from the stack" as soon as its body is pushed. This behavior is useful for potentially *tail-position subexpressions*: after we reduce a `let` block to its body, the `let` block itself can be disposed of, saving stack space.

[`call`$_e$][3] just consists in replacing the `call` expression on the top with its holed counterpart (with an immaterial local environment), and pushing actuals on top of it, so that they will be evaluated starting from the leftmost one, all in the same local environment of the call. When actuals are evaluated the `call` *contractive rule* has the opportunity to fire, provided that the value stack contains a topmost activation with exactly as many 1-dimension bundles as the (current!) number of parameters of the called procedure, and of course provided that a procedure with the appropriate name exists. If that is the case the holed `call` expression is replaced with the procedure body, and the local environment with an environment containing *only* the parameter bindings. It is crucial here *not* to extend the call-time local environment, since we want to prevent nonlocal visibility, for efficiency reasons. In a similar vein to the `let` case, a tail-position holed `call` is *replaced* by the called procedure body.

[`primitive`$_e$] and [`primitive`$_c$] are very similar to their `call` counterparts; the contractive rule cannot fire if the primitive name is not bound, or the primitive function is undefined. Notice that the primitive function is allowed to return a new global state, and the contractive rule effectively establishes it for the resulting configuration.

[`if`$_e$] simply replaces the topmost expression with a holed conditional, pushing the discriminand subexpression on top of it; when the discriminand is completely evaluated, either of the two `if` *contractive rules* [`if`$_c^{\in}$] and [`if`$_c^{\notin}$] may fire, provided the discriminand yielded a 1-dimension bundle: if the value belongs to the conditional case set, the `then` subexpression replaces the holed `if`; otherwise, the `else` subexpression does. The conditional expression is replaced by one of the branch subexpressions without consuming stack space, which is useful in tail contexts.

[`bundle`$_e$] resembles [`call`$_e$] and [`primitive`$_e$]; again the empty environment associated to the `bundle` holed expression is immaterial. [`bundle`$_c$], if the correct number of 1-dimension bundles is on the top of the value stack, replaces them all with a single bundle holding all the values.

[`fork`$_e$] is essentially identical to [`call`$_e$], [`primitive`$_e$] and [`bundle`$_e$]. [`fork`$_c$] is more interesting: if the actual parameter result bundles are 1-dimensioned and correct in number, they are simply replaced by one *future* on the value stack, and the `fork` evaluation terminates immediately: the concurrent evaluation will take place asynchronously in a new thread created for the purpose, and associated to the future identifier in the future state environment. Notice that the thread identifier is also visible to the new thread as the zeroth parameter, to be used in personalities for "`self-thread-name`" forms or thread-local variables.

[`join`$_e$] replaces the topmost expression with its holed counterpart, pushing its

[3] Trivial error fixed in 2014: the original text mistakenly said "[`let`$_c$]" instead of "[`call`$_e$]".

future expression over it; [$\texttt{join}_c$], provided that a 1-dimension bundle is on the top of the stack, that the bundle contains a future value *and the thread corresponding to the future terminated*, returns the result from the thread. [$\texttt{join}_c$] cannot fire until the asynchronous thread has terminated.

[||], provided that that a configuration obtained by making a background thread the foreground thread *could* reduce, allows to perform the reduction "concurrently", in the future state environment.

It is easy to see that in the contractive rules [$\texttt{call}_c$], [$\texttt{primitive}_c$], [$\texttt{bundle}_c$] and [$\texttt{fork}_c$] the local environment associated to the holed expression will in practice always be empty, for reachable configurations.

The role of "ϱ" value separators should be clear at this point: the values of the same bundle are stored on the value stack sequentially, *without* any separators in between — and again in reverse order, because of the LIFO strategy. Separators help establish the correct conditions for a rule to fire, so that *no bundle of the wrong dimension can be used*.

The motivation for activation separators "‡" is similar but slightly more subtle: the problem is being able to distinguish a local, temporary bundle from a surrounding bundle which is being built on the value stack. Without such explicit markers it would be possible to pop the "wrong" number of values from the value stack.

Moreover a procedure can be *redefined*, or even defined for the first time, by one of its actual parameters. We only define semantics if the number of the passed parameters is correct, but their good number cannot be determined until[4] all of them have been evaluated: hence, before letting a contractive rule fire, we have to check that the topmost objects in the value stack be *all and only* the actual values.

At this point it may be worth to remind the reader of Implementation Note 2.2: markers do not necessarily need to be represented and checked for at run time in an efficient implementation; quite the opposite, by specifying that some case yields an error we free ourselves from any implementation constraint.

For this reason we intentionally let, for example, "wrong arity" be an error condition (§2.5.3) instead of specifying some "fallback behaviour" such as ignoring extra arguments or providing defaults for missing ones: in practice an efficient implementation will need to reserve stack frame slots or registers for return addresses, garbage collection structures or for some other implementation bookkeeping purpose: of course passing the wrong number of parameters will likely interfere with these conventions. We do not want this to be made more difficult or less efficient just because of the need of implementing a specific behavior, whose utility was dubious in the first place.

Unfortunately an implementation cannot let configurations grow to an arbitrary

[4] 2014 correction: the original text said "after" instead of "until".

complexity:

Implementation Note 2.7 (dynamic execution resource limits) *Each instance of the following items occupies some memory in an implementation (See Implementation Note 2.3):*

- *a stack item;*
- *a value stack item;*
- *a local environment binding.*

Some implementations may further limit the stack item and value stack item number to another smaller constant, independently from memory usage. □

2.5.2 Sequential reduction

As we have just remarked in §2.5.1, we value the predictability of ε_0 semantics, with its well-specified evaluation strategy. In the same vein *determinism* in an evaluation relation is a desirable property.

It is easy to observe that, save for the parallel rule, the reduction relation *is* in fact trivially deterministic, up to the (immaterial) choice of thread identifiers.

Definition 2.8 (sequential small-step reduction) *We define the sequential small-step evaluation relation* $\_\_\_ \longrightarrow^{\not\parallel}_{\mathbb{E}} \_\_\_ \subseteq (\mathbb{S} \times \mathbb{V} \times \mathbb{\Gamma}) \times (\mathbb{S} \times \mathbb{V} \times \mathbb{\Gamma})$ *according to the rules on pp. 32-33, minus the parallel rule.* □

Interestingly, a sequential reduction can still work with `fork` and `join`, and futures can be passed around and even created anew or joined if their result is ready: since the only source of non-determinism is the actual concurrent reduction, as long as no background thread "advances" it is possible to work with futures using only $\_\_\_ \longrightarrow^{\not\parallel}_{\mathbb{E}} \_\_\_$.

2.5.3 Failure

In §2.5.1 we have explicitly shown that there are cases in which the small-step semantics is undefined because rule premises cannot be satisfied. After having formalized the notion of "correct reduction", here we are going to exactly specify and classify failure conditions.

Definition 2.9 (error configurations) *We define the error configuration relations fails because of environments, written as "*$\_ \_ \_ \longrightarrow_{\mathbb{E}} \text{※}_{\not\mathbb{X}}$*" and fails because of dimension, written as "*$\_ \_ \_ \longrightarrow_{\mathbb{E}} \text{※}_{\#}$*", all subsets of the set of configurations* $\mathbb{S} \times \mathbb{V} \times \mathbb{\Gamma}$*, by the following rules:*

$$\frac{}{(x_h,\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} \text{※}_{\not\mathbb{X}}}\ x \notin dom(\Gamma_{\texttt{global-environment}}[\rho])$$

$$\frac{}{([\texttt{let } x_1...x_n \texttt{ be } \square \texttt{ in } e_{h_2}]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}} \nexists m : (m \geqslant n \wedge V \equiv\ \wr c_m c_{m-1}...c_2 c_1 \wr V')$$

$$\frac{}{([\texttt{call } f\ \square]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}} \neg(\Gamma_{\texttt{procedures}} : f \mapsto (x_1...x_n, e_h) \wedge V \equiv\ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr \ddagger V')$$

$$\frac{}{([\texttt{primitive } \pi\ \square]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}} \Gamma_{\texttt{primitives}}(\pi) :_{\#} n \to m \wedge V \not\equiv \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr \ddagger V'$$

$$\frac{}{([\texttt{if } \square \in \{c_1...c_n\} \texttt{ then } e_{h_2} \texttt{ else } e_{h_3}]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}} V \not\equiv\ \wr c \wr V'$$

$$\frac{}{([\texttt{bundle } \square]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}} V \not\equiv \wr c_1 \wr c_2 \wr ... \wr c_{n-1} \wr c_n \wr \ddagger V'$$

$$\frac{}{([\texttt{fork } f\ \square]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}} \neg(\Gamma_{\texttt{procedures}} : f \mapsto (x_0...x_n, e_h) \wedge V \equiv\ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr \ddagger V')$$

$$\frac{}{([\texttt{join } \square]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}} V \not\equiv \wr c \wr V'$$

The fails because of a primitive relation $\_\ \_\ \_ \longrightarrow_{\mathbb{E}} ※_{\mathbb{P}} \subseteq \mathbb{S} \times \mathbb{V} \times \mathbb{\Gamma}$ *is a superset of the relation defined by the following rule:*

$$\frac{}{([\texttt{join } \square]_{h_0},\ \rho).S\ \wr c \wr V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\mathbb{P}}} c \not\equiv \mathcal{T}(t)$$

The exact definition of $\_\ \_\ \_ \longrightarrow_{\mathbb{E}} ※_{\mathbb{P}}$ *relies on the specific set of available primitives, which we intentionally leave open.*

We define the generic "fails" relation, written as "$\_\ \_\ \_ \longrightarrow_{\mathbb{E}} ※$*", as the union of the specific failure relations:* $(\_\_\_ \longrightarrow_{\mathbb{E}} ※) = (\_\_\_ \longrightarrow_{\mathbb{E}} ※_{\mathbb{X}}) \cup (\_\_\_ \longrightarrow_{\mathbb{E}} ※_{\mathbb{P}}) \cup (\_\_\_ \longrightarrow_{\mathbb{E}} ※_{\#})$*. We also call final failure configuration a configuration that fails. A final configuration is either a final success configuration or a final failure configuration.* ▫

Since the definition above does not mention background threads at all we have that failure in a background thread does *not* propagate to any other thread. We chose this solution in the interest of simplicity and realism for a core language such as ε_0, which should reflect the behavior of system-level facilities. Of course higher-level personalities are free to implement more complex policies, as hinted at in §2.5.4.

Since failure never propagates to other threads in ε_0, there is no need for alternate "sequential" relations for failure.

As a further point of note, above we have chosen to classify the failure of `join`-ing an object different from a future as a primitive error, because of the strong analogy of the condition with a primitive "wrong parameter" error[5].

Many actual primitives also fail for some values of their parameters, even when they receive the correct number of them. For example a division primitive "$\div$" might fail on a zero divisor; writing "$\mathcal{N}(0)$" for zero as in §2.2, we get:

$$\frac{}{([\texttt{primitive} \div \Box]_{h_0},\ \rho).S\ \wr\mathcal{N}(0)\wr c\wr\ddagger V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\mathbb{P}}}$$

We intentionally omit a list of all the specific cases of primitive failure, a complete specification belonging in the primitive definition — with the only constraint of having failure rules covering *all* possible failure cases; in other words, given a set of parameters a primitive either fails or returns a result but no other behavior such as divergence is possible, as specified by Axiom 2.10, which we are now ready to state:

Axiom 2.10 (primitive "totality") *For any primitive π such that $\Gamma_{\textit{primitives}}(\pi) :_{\#} n \to m$ in some state Γ and for each sequence $\langle c_1, ..., c_n\rangle$, exactly one of the following holds:*

- *there exist $c'_1, ..., c'_m, \Gamma'$ such that $\Gamma_{\textit{primitives}}(\pi)(c_1, ..., c_n, \Gamma) = \langle c'_1, ..., c'_m, \Gamma'\rangle$;*
- *for any h_0, S, V we have $([\texttt{primitive}\ \pi\ \Box]_{h_0},\ \rho).S\ \wr c_1\wr ...\wr c_n\wr\ddagger V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\mathbb{P}}$.* ▫

We can prove a result in the same spirit for general expressions: any given configuration either can be reduced for at least one more step, or it immediately fails; but it is not possible that a non-failing configuration does not allow reductions (unless joining a future), or that a configuration simultaneously fails and allows a sequential reduction:

Proposition 2.11 (reduce xor fail xor wait) *Given any reachable configuration $\chi = (e_h, \rho).S\ V\ \Gamma$ we have exactly one of the following:*

- *there exist S', V' and Γ' such that $(e_h,\ \rho).S\ V\ \Gamma \longrightarrow^{\nparallel}_{\mathbb{E}} S'\ V'\ \Gamma'$;*
- *$(e_h,\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※$;*
- *there exist $t \in \mathbb{T}$, $V' \in \mathbb{V}$ such that $\chi = ([\texttt{join}\ \Box]_h,\ \rho).S\ \wr\mathcal{T}(t)\wr V'\ \Gamma$.*

Proof (sketch) Since the main stack is not empty, χ is not terminal.

We are dealing with $\_\_\_ \longrightarrow^{\nparallel}_{\mathbb{E}} \_\_\_$ rather than $\_\_\_ \longrightarrow_{\mathbb{E}} \_\_\_$, hence [||] cannot fire, by Definition 2.6.

In any configuration where the top expression is an ε_0 non-holed expression except for the variable, we may apply an expansive rule or [*constant*] for $\_\_\_ \longrightarrow^{\nparallel}_{\mathbb{E}} \_\_\_$, leading to an evaluation step: all such rules can always fire independently of subexpressions, the state of the environments or the stacks (Definition 2.6).

[5] It is easy at this point to mistake such an error for a type error. The difference is actually subtle, and will be dealt with in §3.3.1

If $\chi = ([\texttt{join}\ \square]_h,\ \rho).S\ \wr\mathcal{T}(t)\wr V'\ \Gamma$ for some t, V' then the thesis follows trivially.

In the remaining cases the top expression is a variable or a holed expression, and another disjoint set of rules applies. Then one of the contractive rules for $\_\_\_ \longrightarrow^{\not\parallel}_{\mathbb{E}} \_\_\_$, or an error rule for $\_\_\_ \longrightarrow_{\mathbb{E}} \divideontimes_{\not\chi}$, $\_\_\_ \longrightarrow_{\mathbb{E}} \divideontimes_{\#}$ and $\_\_\_ \longrightarrow_{\mathbb{E}} \divideontimes_{\mathbb{P}}$ in Definition 2.9 must apply.

In all the cases above it is easy to see that each configuration matches the premise of exactly one rule (in particular, since χ is reachable, the value stack must have $\wr$ on top); the only nontrivial case is $[\texttt{primitive}\ \pi\ \square]_h$, in which either $[\texttt{primitive}_c]$ or a $\_\_\_ \longrightarrow_{\mathbb{E}} \divideontimes_{\mathbb{P}}$ rule applies because of Axiom 2.10. ■

2.5.4 Error recovery and personalities

At the level of ε_0 all errors are fatal. In the interest of simplicity and efficiency, no mechanism is provided for handling a case of failure by recovering or retrying. Any such machinery can be defined in high-level personalities by checking for failure conditions at run time with explicit conditional expressions to be automatically generated; in this way it is possible to completely prevent ε_0 failures from ever occurring, if so desired.

As usual and in the same spirit of typing, the personality implementor has the freedom of choosing an efficient model where failures are always fatal, or a friendlier alternative where the personality presents an ordinary ε_0 configuration as an "error" state from which the conventionally "normal" execution can resume.

Like for static typing it is also possible to check for the possibility of failures "statically" at code generation time, and generate fast code under the assumption that some kind of failure is impossible. The next chapters hint at how one can define such analyses.

2.6 One-step dynamic semantics

When dealing with "toplevel" ε_0 expressions, often we are less interested in the small-step evaluation relation $\_\_\_ \longrightarrow_{\mathbb{E}} \_\_\_$ than in its *iteration*: where only the final configurations (if any) are of interest it is convenient to completely ignore stacks and value stacks, restricting our attention to an expression, an initial state, its result bundle and the terminal state.

Definition 2.12 (one-step convergence) *We define the one-step operational semantics relation for expressions* $\_\_ \Downarrow_{\mathbb{E}} \_\_ \subseteq (\mathbb{E} \times \mathbb{\Gamma}) \times (\mathbb{C}^* \times \mathbb{\Gamma})$ *by the rule:*

$$\frac{(e_h, \varnothing)\ \wr\ \Gamma \longrightarrow^{+}_{\mathbb{E}} \langle\rangle\ \wr c_n ... c_1 \wr\ \Gamma'}{e_h\ \Gamma \Downarrow_{\mathbb{E}} \langle c_1 ... c_n \rangle\ \Gamma'}$$

Similarly, we define the one-step sequential operational semantics relation for expressions $\_\_ \Downarrow^{\not\parallel}_{\mathbb{E}} \_\_ \subseteq (\mathbb{E} \times \mathbb{\Gamma}) \times (\mathbb{C}^* \times \mathbb{\Gamma})$ *by the rule:*

$$\frac{(e_h, \varnothing)\ \wr\ \Gamma \longrightarrow_{\mathbb{E}}^{\nparallel +} \langle\rangle\ \wr c_n ... c_1 \wr\ \Gamma'}{e_h\ \Gamma \Downarrow_{\mathbb{E}}^{\nparallel} \langle c_1 ... c_n \rangle\ \Gamma'}$$

When we have that "$e_h\ \Gamma \Downarrow_{\mathbb{E}} \langle c_1 ... c_n \rangle\ \Gamma'$" we say that e_h in Γ converges to $\langle c_1 ... c_n \rangle$ in Γ'. In the same way when we have "$e_h\ \Gamma \Downarrow_{\mathbb{E}}^{\nparallel} \langle c_1 ... c_n \rangle\ \Gamma'$" we say that e_h in Γ sequentially converges to $\langle c_1 ... c_n \rangle$ in Γ'.

We may omit state names or results when irrelevant in context. ▫

It trivially follows from the determinism of $\_\_\_ \longrightarrow_{\mathbb{E}}^{\nparallel} \_\_\_$ that $\_\_ \Downarrow_{\mathbb{E}}^{\nparallel} \_\_$ is also a (partial) *function.*

Notice that according to our definition a reduction chain of $\_\_\_ \longrightarrow_{\mathbb{E}} \_\_\_$ may converge even if some background thread potentially runs forever, when a finite reduction chain *exists* for $\_\_\_ \longrightarrow_{\mathbb{E}}^{\nparallel} \_\_\_$, and hence also for its super-relation[6] $\_\_\_ \longrightarrow_{\mathbb{E}} \_\_\_$.

It is also useful to speak of the *eventual* failure of an expression in a state, ignoring the zero or more reduction steps leading to the failure configuration, and the specific failure configuration as well:

Definition 2.13 (eventual failure) *For each error-configuration relation $\_\_\_ \longrightarrow_{\mathbb{E}} ※_f$, with $f \in \{\text{“”}, \text{“}\mathbb{X}\text{”}, \text{“}\mathbb{P}\text{”}, \text{“}\#\text{”}\}$, we define a corresponding eventual failure relation $\_\_ \Downarrow_{\mathbb{E}} ※_f \subseteq \mathbb{E} \times \mathbb{\Gamma}$ by the meta-rule:*

$$\frac{(e_h, \varnothing)\ \wr\ \Gamma \longrightarrow_{\mathbb{E}}^{*} S\ V\ \Gamma' \qquad S\ V\ \Gamma' \longrightarrow_{\mathbb{E}} ※_f}{e_h\ \Gamma \Downarrow_{\mathbb{E}} ※_f}$$

If we have that "$e_h\ \Gamma \Downarrow_{\mathbb{E}} ※_f$" with $f \in \{\text{“”}, \text{“}\mathbb{X}\text{”}, \text{“}\mathbb{P}\text{”}, \text{“}\#\text{”}\}$ we respectively say that e_h in Γ eventually fails, eventually fails because of environments, eventually fails because of primitives, or eventually fails because of dimension. ▫

Finally, we characterize *looping* expressions and states:

Definition 2.14 (divergence) *We define the divergence relation $\_\_ \Uparrow_{\mathbb{E}} \subseteq \mathbb{E} \times \mathbb{\Gamma}$ the following way:*

let e_h be an expression and Γ be a state; then we say that e_h diverges in Γ and we write "$e_h\ \Gamma \Uparrow_{\mathbb{E}}$" if for any configuration χ such that $(e_h, \varnothing)\ \wr\ \Gamma \longrightarrow_{\mathbb{E}}^{+} \chi$ there exists another configuration χ' such that $\chi \longrightarrow_{\mathbb{E}} \chi'$. ▫

We defined divergence with the parallel reduction relation $\_\_\_ \longrightarrow_{\mathbb{E}} \_\_\_$ rather than its sequential restriction; hence our notion of divergence covers both "busy looping" in the foreground and waiting forever for a background thread.

It may be worth to stress how, for example, the sentence "e_h in Γ does not converge" has a different meaning from "e_h in Γ diverges", since it is possible that e_h in Γ eventually fails. The same problem holds for the phrases "converges" and "eventually fails".

We will avoid such wording in the negative.

[6] Error fixed in 2014: the original text mistakenly said "sub-relation" instead of "super-relation".

2.7 Summary

We conceived the ε_0 core language for expressivity, efficiency and ease of formal manipulation: ε_0 is powerful but idiosyncratic and unsuitable for direct use by human users, who are expected to only access it through extensions.

After dealing at length with design issues and providing a rationale for ε_0 language we proceeded to formally specify its syntax, semantics and error conditions.

We also described a sufficient set of conditions under which an implementation is compelled to respect the specified behavior, allowing for both *inefficient but friendly* and *efficient but unsafe* implementations.

The small-step operational semantics is relatively simple and has a deterministic sub-relation obtained by simply ignoring one rule.

We defined the "one-step" semantics, hiding the complexity of stacks, by iterating the small-step reduction relation.

Chapter 3

Reflection and self-modification

The presentation of ε_0 in §2 showed the *state* component as already containing procedure and global bindings, but did not illustrate any explicit way of updating either.

In this chapter we will start by discussing global definitions, and then proceed to clarify what we mean by a "program"; our somewhat unusual solution has important implications on how programs are loaded, saved, and compiled.

Contents

3.1 Global definitions

The expression semantics given in §2.6 does not explicitly mention any functionality to alter the set of procedure or global bindings that we have always considered as *already* defined, as part of the state; anyway such functionality is clearly needed: if not for anything else, at least for defining new recursive procedures, as expressions can not express recursion without referring global procedures; and, of course, global definitions are useful for reasons of modularity.

A "traditional" solution to this problem would consist in adding *toplevel forms* to ε_0 as a new syntactic category: toplevel forms would comprise a procedure definition form and a non-procedure definition form; the program would then become a sequence of toplevel definitions, possibly followed by a main expression returning a final "result".

We have several good reasons to reject this simplistic notion of a fixed program to be written from start to finish and then executed:

- in the spirit of Forth[1] and Scheme, we want to also support *interactive* systems interleaving user input with evaluation and answers;

[1] It is not by coincidence that we mention Forth first in this case. One important lesson of Forth is how complex programs can be written even with no support whatsoever for typing, *provided that* each small component is individually testable. We extrapolate the following motto

- having *exactly two* toplevel definition forms may be adequate for ε_0, but certainly not for its extensions: a personality may need global definitions for new entities such as classes, exceptions, or types. Even syntax is not fixed: we want to make new entities and their associated syntactic extensions definable at any point during the user interaction, to be immediately available for use;

- adding toplevel forms to ε_0 is not necessary, since expressions are already powerful enough to express state updates using primitives or procedures;

- a powerful language should let the user update global definitions *from any program point*, not just at the top level.

For all these reasons we will simply assume the presence of the *global-definition* or *self-modification* procedures `state:global-set!` and `state:procedure-set!`, to be defined later in §5.4.1.3, p. 95. We also assume the presence of their companion *reflective procedures* `state:global-get`, `state:procedure-get-formals`, `state:procedure-get-body`, `state:global-names` and `state:procedure-names`.

No toplevel forms are needed[2]: definitions are expressions like any others, and can be executed at the top level or just as well within other expressions.

3.2 Programs and self-modification

The last point in the dotted list above, easily the most controversial, illustrates well the tension between our will of providing an expressive system, and the desire of also keeping the language easy to reason about and efficient — in mainstream terms, the *dynamic vs. static* debate.

At a first look ε_0 with global-definition procedures appears flatly sided with the "dynamic" party, allowing any program to capriciously modify itself at run time; for example in §2.5.1 at p. 36 we even considered the case of calling a not-yet-existing procedure which is *created by one of its actual parameters*. Indeed, we will find use for creating procedures from arbitrary expressions (§5.4.4.4); but whenever possible we would still prefer not to renounce to better intellectual manageability, and efficiency.

As a reasonable compromise and a way out of the dilemma, it is possible *to freely use global-definition procedures to let the program reach a final "static" form*; after that point, the program may be analyzed to check for properties and compiled efficiently, under the assumption that no more self-modifications will occur.

---

from our experience of working with ML, Lisp and Forth: *the less typing, the more important a Read-Eval-Print Loop.*

[2] Global-procedure-definition procedures are not particularly friendly to use directly, since the user has to pass *expressions* as parameters, and expressions are relatively complex data structures which need to be built. However a friendly definition form is not hard to define on top of global-definition procedures, by using a macro (§5).

3.2.1 Programs

The most convenient notion of a program, for our purposes, is slightly unusual. Given a state and an expression, we can imagine to somehow generate a snapshot containing the "frozen" state, plus the expression.

"Executing" a program then means to fire up evaluation on the saved expression from the resumed state:

Definition 3.1 (program) *Let* Γ *be an* ε_0 *state and* e_h *be an* ε_0 *expression; then we define their corresponding program as the pair* $(\Gamma[^{\varnothing}_{\texttt{futures}}], e_h) \in \mathbb{\Gamma} \times \mathbb{E}$. ▫

We intentionally disregard the background threads in $\Gamma_{\texttt{futures}}$, in order not to have to deal with execution stacks or partial expression evaluation. Background threads do not look particularly useful anyway in this context, since the main idea is simply to use global-definition procedures to have the program self-modify into something which contains every needed auxiliary procedure and data, before the "main expression" can be finally evaluated; clearly, the main expression itself will be free to create background threads.

Of course there is no guarantee that a program, when executed, will not start self-modifying again.

3.2.2 Static programs

A static program is a program which, when executed, never self-modifies:

Definition 3.2 (static program) *Let* (Γ, e_h) *be a program; then we say that it is semantically static or static if in no configurations reachable from evaluating* e_h *in* Γ*, the global environment or the procedure environment are different from the ones in* Γ*.*

More formally, a program (Γ, e_h) *is static if for all* $(S'\ V'\ \Gamma')$ *such that* $(e_h, \varnothing) \wr \Gamma \longrightarrow^{+}_{\mathbb{E}} (S'\ V'\ \Gamma')$ *we have that* $\Gamma_{\texttt{global}} = \Gamma'_{\texttt{global}}$ *and* $\Gamma_{\texttt{procedures}} = \Gamma'_{\texttt{procedures}}$. ▫

We consider re-defining a global to be self-modification: anyway the user can still define mutable variables in the style of ML and Forth in a static program, by adding one indirection level so that a global maps to a memory cell, whose content can be updated any number of times without affecting its identity. The value of imperative variables would then be held in the *memory* state environment (§2.4.3), which does not affect staticity.

Interestingly, the use of *reflective* procedures is not problematic even for a static program: a static program can safely *read* its own global and procedure state environments, which are constant by definition.

Our *semantic* versus *syntactic* naming convention is important and deserves some comments. The convention comes from standard garbage collection jargon [98], and is used to distinguish between a datum which will not be accessed in the rest of the computation from a datum which can not be reached by traversing pointers

from the roots. It is undecidable whether a heap object is "semantic garbage", so garbage collection works by recycling "syntactic garbage", a conservative decidable approximation (any piece of semantic garbage is also syntactic garbage).

Like the property of being semantic garbage our semantic staticity property is trivially undecidable, so it would be tempting to define a notion of "syntactic staticity" involving the use of global-definition procedures in reachable code. Such attempts are doomed to fail in ε_0, because of the way global-definition procedures are defined (§5.4.1.3): in practice, at least with our current set of primitives[3], it is always possible to modify procedures or globals with ordinary memory stores, bypassing the "high-level" procedures for program self-modification.

Syntactic staticity properties *are* definable in typed personalities, where dynamic or static checks prevent the user from writing at arbitrary memory addresses.

In accordance with our open-ended design principles we do not consider an "error" for a program to be self-modifying; yet staticity remains desirable since self-modification makes most analyses impossible, prevents many compiler optimizations including inlining, and indeed challenges the very idea of compilation[4].

A static program can instead be compiled and optimized in a traditional way and as a consequence of our design a *whole-program* approach, lending itself to global optimizations [94], feels particularly natural. Since the expressions occurring in a fixed program are all known, it is easy to build global tables with handles as keys (§2.1.3), to perform any kind of analysis[5].

Particularly in an untyped context, where users are supposed to be competent, it is reasonable to consider a certain program as semantically static *when users demand so* by requesting to analyze or compile a program in a modality which takes advantage of staticity. We stress once more how all such functionality, including compilers, can be written in the language itself as part of a "library", and is not specially hardwired in the system in any way[6].

[3] It would be possible to bootstrap the language with a different set of primitives (§5.4.1.3), so that `state:global-set!` and `state:procedure-set!` are primitives themselves and do not depend on others. However such a solution would be unrealistic for a practical implementation, where identifiers and expressions are data structures like any others.

As a slightly more subtle point, a syntactic staticity guarantee would also have to prevent *destructive modification of expressions*, which is possible as well in our implementation of §5.

[4] It *is* possible to compile only parts of the code, as several Lisp systems do, but the interaction between interpreted code and compiled code complicates design, also requiring dynamic invalidation and substitution of compiled code. It is also possible to have a JIT, or more simply a compiler to be executed at run time which translates every expression as soon as it is generated, like in the SBCL Common Lisp system [73].

[5] A useful notion for which we cannot claim novelty: the idea of attaching user-defined data to syntactic objects, now mostly popular because of Java "annotations" (`http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`), was already quite explicit in McCarthy's 1959 LISP [51].

[6] We did not implement a complete compiler yet (§5.4.5) but we have a custom bytecode virtual machine, and the beginnings of native bindings. No fundamental obstacle in implementing an ε_0 native compiler is apparent, and we plan to write one in the coming months.

Since the set of procedures in a static program is fixed and so is its main expression, it makes sense to define a notation to show a program in a "linear" (and leaner) form. It is also convenient to speak about individual program components using the "$\_ \in \_$" operator, without making state environments explicit.

This will be useful in §4, when we describe a static analysis in detail.

Definition 3.3 (static program linear syntax) *Let $p = (\Gamma, e_h)$ be a static program. If $\Gamma_{\texttt{procedures}} = \{f_1 \mapsto (x_{1_1}...x_{1_{n_1}}, e_{h_1}), f_2 \mapsto (x_{2_1}...x_{2_{n_2}}, e_{h_2}), ..., f_m \mapsto (x_{m_1}...x_{m_{nm}}, e_{h_m})\}$, then we can write the whole program as:*
"[`procedure (`$f_1\ x_{1_1}...x_{1_{n_1}}$`)` e_{h_1}]
[`procedure (`$f_2\ x_{2_1}...x_{2_{n_2}}$`)` e_{h_2}]
...
[`procedure (`$f_m\ x_{m_1}...x_{m_{nm}}$`)` e_{h_m}]
e_h".
We also write:

- "[`procedure (`$f\ x_1...x_n$`)` e_{h_1}] $\in p$" *to mean that* $\Gamma_{\texttt{procedures}} : f \mapsto (x_1...x_n, e_{h_1})$;
- "$e_h \in p$", *to mean that* e_h *is the main expression of* p. □

3.2.3 When to run analyses

In traditional languages the act of performing a "static" analysis means running some procedure over the syntax trees from a compilation unit *before* the unit is compiled or executed; but with our program notion above blurring phases and units, the very idea of "static analysis" in the context of ε becomes fuzzy.

Activities closely analogous to static analysis remain meaningful: for example in a statically-typed personality the procedures and global variables which are part of a program at a given point can still be usefully checked for type safety, *independently from the way each entity was defined* in the past evaluation history.

Some analyses may be attempted even for non-static programs. The problem becomes rather *the point in time* at which to run analyses: since no "end point" is apparent, no obvious solution comes to mind. A personality might run some or even all the analyses right after evaluating *each* toplevel expression; as a more radical hypothesis an advanced editor such as Emacs can certainly be programmed to communicate with an interpreter after each *character* modification, demanding to run analyses[7] and visualizing their continuously updated results.

Of course any similar solution needs to cope with the possibility of yet unresolved forward references, which cannot be prevented in general due to the mutual

[7]The difficulty of this approach is due less to program analysis than to the difficulty of defining the semantics of incremental modifications to non-contiguous points of a program. This seems hard to accomplish for ε without rebuilding the entire state from scratch at every change, at a prohibitive cost in performance; caching mechanisms can be conceived for some personalities to make such operations more efficient.

recursion inherent in ε_0 procedures: it is to be expected and regarded as normal that analyses fail at some points where the current set of global definitions is "open".

One likely appropriate time for running analyses is right before compilation, since no unresolved forward references would be present at that point; but in ε compilation does not necessarily mark any "terminal" point of the evaluation history, either. It seems reasonable to also allow analyses to be run at any point, *on demand*.

We will see in §5.4.1.5 how transforms may be conveniently used to automatically associate analyses to global definitions.

3.3 Unexec

Our programs, be they static or not, are in fact "system images", which would be convenient to write to disk for later execution, or even to be transferred to different computers; the main expression to be saved as part of the program might simply be *a call to the REPL procedure, itself calling the interpreter*: that way restoring the system image would open an interactive session in the saved state.

One could even envisage "snapshots" as a way of saving the current system state before performing an experimental and potentially destructive modification in an interactive way: if the modification fails, the user can revert to the old state by loading the last snapshot, presumably a much faster operation than re-building the previous state by repeating the same self-modifications which generated it in the first place from the initial state.

The functionality cursorily described above resembles the Emacs "unexec" hack [47, §Building Emacs]. Emacs consists of a relatively small Lisp interpreter written in C which contains the core primitives, plus the bulk of the system implemented in Lisp; in order to avoid loading hundreds of Lisp files at every startup, the native Emacs executable is built so that it fires up *in the state which would be produced by loading the initial Lisp files*. Building such a functionality in C with native processes is tricky and requires system-specific low-level code.

Despite the similarity of intent our implementation will be much simpler, and largely machine-independent.

The general idea of our unexecing strategy is to simply *marshal* data structures into a linear representation which can later be read back in an *exec* phase, based on unmarshalling.

The composition of unexec and exec yields a state identical[8] to the original one up to buffer addresses.

[8] We are assuming that memory encodes the complete global state, but this assumption breaks if the state refers system structures such as open files or sockets: unexec can not reproduce any object out of its process address space.

We cannot claim novelty for this idea, considering for example Hoare's early intuition in [36, §3.3(2-3)]; in a couple recent systems unexecing exists, but plays a less central role than in ours: SML/NJ for example supports a "`heap2exec`" utility [49]; it generates native code,[9] yet it can only run on a couple of platforms — `heap2exec` is not by any means "the compiler" for SML/NJ, but rather just one tool among others. Unexec support has also been discussed or experimented with for Perl, Python and Guile [21].

3.3.1 The stuff values are made of

Since any realistic implementation must work on general-purpose Von Neumann machines, it is clear that the implementations of all state environments and expressions share in practice *the same machine memory*; and that memory holds the data structures we have to marshal.

Encoding details will be made clear in §5.4.1.3; but even without specifying here how each kind of data is represented, we need to describe the *memory model* followed by all our in-memory objects. As a consequence of other design decisions, the actual data structures to be marshalled for unexecing will be surprisingly few in number (§5.4.2).

We can ignore background threads, which are not involved in unexecing as they are excluded from programs as per Definition 3.1. The remaining values are of only two kinds:

- *unboxed* values;
- heap buffer *pointers*, also called *boxed* values.

A machine word, in practice not wider than a general register (32 or 64 bits on modern machines), can hold either an unboxed value, or a pointer to a buffer; a buffer is a contiguous array of other machine words in heap memory. Pointers are always *initial*: we preferred to simply avoid *interior* pointers as they may exhibit bad interactions with some garbage collection algorithms which we may want to adopt in the future [98], even if ours has no such restriction (§6.3.2).

Unboxed values are often used for *fixnums*, which is to say fixed-range integers, represented in two's complement on modern hardware. Booleans, characters and enumerates also fit comfortably in the range of unboxed values. No provision is made for objects smaller than a word: at this level, one word is the smallest representable datum. Complex data containing multiple "fields" usually need to be boxed, but if all fields put together fit within the width of an unboxed datum, they can also be packed into a single word: from the point of view of the memory model there is no difference between a single-field and a multi-field unboxed object. Efficient

[9] Keeping the two functionalities separate has the advantage of providing a working unexec feature also on platforms where a native compiler is not implemented.

implementations of dynamically-typed personalities are free to reserve some bits as tags in unboxed objects and pointers [35], in the case of pointers exploiting the fact that allocation alignment will free at least two or three bits for all buffer addresses, on current byte-addressed machines (§6.3).

Some values are in practice necessarily boxed, notably reified *expressions*; as a consequence in some cases what we informally called a "value" in §2 is actually a function of the pointer, which we use as a reference to the entire object to pass around, plus all the memory which it refers, closed under the "points-to" relation.

Values can then be visualized as a graph, possibly containing converging edges and cycles.

It is in practice possible to alter memory to change a boxed datum component even when such value does not appear as "mutable" in the semantics, for example by using a store primitive on an expression datum. Such practices would entail primitive failure[10] and hence not respect the hypotheses for implementation guarantees (see Implementation Note 2.2); the fact that they are possible does *not* constitute a violation of ε_0 semantics.

The following linearized textual format for memory data structure including addresses is very convenient for debugging the implementation and also as a generic fallback "untyped printer":

Syntactic Convention 3.4 (memory dump) *We dump a given datum into a string of colored characters, according to its shape. There are two cases:*

- *an unboxed datum is written in green in decimal, as a two's complement signed integer;*
- *a pointer is written as a hexadecimal number, prefixed by the string "*`0x`*";*
  - *if the referred buffer occurs for the first time in the data structure (depth-first left-to-right), its address is written in red followed by a dump of its buffer elements between brackets, with consecutive content words separated by a space;*
  - *if the referred buffer has already occurred, its address is written in yellow and the buffer content is not repeated.* □

With some practice it is not difficult to make sense of quite complicated data structures by reading memory dumps. Despite not being strictly needed to parse textual

[10] We did not specify explicit rules for our chosen set of primitives, including preconditions to be satisfied to avoid failure; however we can quickly hint at a solution: we can imagine that each buffer contains an initial boolean tag word, recording its mutability or lack thereof; the store primitive will only permit to write mutable buffers, and another primitive will be available to change a mutability tag from mutable to immutable, but never the converse. Load and store primitives would implicitly skip the tag word in the offset they receive (§5.4.1.3).

Of course the implementation does not need to actually represent the mutability tag word: Implementation Note 2.2 permits us to *assume that no failure occurs*, and still respect our specification.

dumps, color makes it easier for humans to recognize a structure's shape at a glance.

In most cases (but not all: see the discussion of hashes in §3.3.2.2) the actual numeric address held by pointers is not relevant for algorithms, and a pointer simply "identifies" a certain buffer, independently from its specific placement in memory. It is hence reasonable to represent data graphically, ignoring addresses and simply using arrows for pointers, multi-slot boxes for buffers, and numbers for unboxed data.

Such "address invariance" is fortunate, since usually we do not have control over buffer addresses at allocation time[11], hence we cannot reliably re-create a buffer at a specified memory address. What the composition of marshalling and unmarshalling will accomplish, then, is the reproduction of the *data structure graph* (Figures 3.1 and 3.2). For example, after marshalling and unmarshalling, the data structure dumped in Figure 3.2 might be "cloned" into `0x26aaaf0[0x2899220[42] 0x3078920[0x2899220 0]]`.

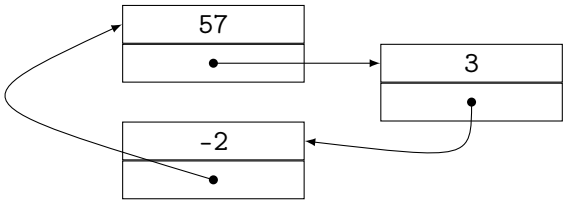

Figure 3.1: A circular list holding the fixnums `57`, `3` and `-2`, whose dump could be `0x27032d0[57 0x279ead0[3 0x28a66e0[-2 0x27032d0]]]`.

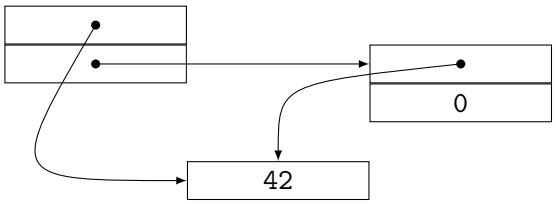

Figure 3.2: An example of sharing: a two-element list using `0` as a terminator whose elements both point to the same one-element buffer, holding the fixnum `42`. One possible dump is `0x29ecd90[0x2714220[42] 0x29549f0[0x2714220 0]]`.

3.3.2 Marshalling

Textual dumps as per Syntactic Convention 3.4 could serve as a marshalling format; however our implementation marshals data structures into binary files, for efficiency

[11] System libraries ultimately choose data structure addresses, providing very few guarantees. Sometimes the problem is made even worse by deliberate address space boundary randomizations performed for security reasons [74].

reasons. Since specific pointer values are immaterial, in marshalling we replace them with sequential 0-based identifiers, which enables some minor optimizations. The logic of marshalling and unmarshalling algorithms resembles moving garbage collecting algorithms such as *semispace* [98], which have to recursively "clone" data structure graphs.

Similarly to textual dumps, when *marshalling*, the idea is to recursively *trace* a data structure keeping into account which buffers we already visited; marshalling produces a sequence of zero or more "buffer definitions" followed by the single main object, be it a pointer or an unboxed value. Pointers are encoded as buffer indices, following the definition order.

Conversely, the *unmarshal* procedure will allocate and fill buffers, and then resolve buffer identifiers into pointers in a second pass.

In our current implementation each file field is a 32-bit big-endian word[12]. A *binary dump* (see Figure 3.3) begins with a word holding the *number of buffers*, followed by the same number of "buffer definitions", and finally by the "main object"; each buffer definition contains a word encoding the *buffer size* in words, following by as many "elements", each element containing two words: either a *0 tag* for an unboxed object followed by the *content*, or a *1 tag* for a boxed object followed by the *buffer index* (in the order of buffer definitions, 0-based); the main object is one further element.

$$\textit{buffer-no}\ \overbrace{(\textit{element-no}\ \underbrace{((0|1)\ \textit{fixnum})^*}_{\textit{element-no}\ \text{times}})^*}^{\textit{buffer-no}\ \text{times}}\ (0|1)\ \textit{fixnum}$$

Figure 3.3: Binary dump file format. Each of *buffer-no*, *element-no*, 0|1 and *fixnum* is encoded as a 32-bit big-endian word.

3.3.2.1 Boxedness tags

Up to this point we have assumed that marshalling, and textual dumping as well, can discriminate between pointers and unboxed objects; but this it not possible at the hardware level.

On the physical machine pointers are memory addresses, which is to say *numbers*, and as such in principle indistinguishable from unboxed objects such as fixnums.

[12]Using 32-bit rather than 64-bit words helps to avoid relying on non-portable behavior by mistake when dumping very large unboxed data. We could reduce tag words to bytes or even single bits, but particularly in the latter case it is not clear whether the denser format would compensate for the additional required shuffling in terms of time efficiency. Space consumption is not a problem: typical unexec binary dumps have sizes ranging in the order of one to a few megabytes.

Modern hardware and operating systems tend to guarantee that objects will not be allocated at very low addresses, so in practice it may be safe to assume that all pointers have a numeric value larger than some constant such as 2^{16} [5]; alignment causes all pointers to be multiples of the word size, possibly times some other small factor; however many large fixnums remain effectively impossible to discriminate from pointers.

The solution is providing a one-bit *boxedness tag* associated to each datum, plus a *dimension field* per buffer — dimensions tending not to be overly problematic in practice[13].

The bit can be stored within each word itself, if reducing its payload width is acceptable; otherwise a *much* less efficient but more flexible solution consists in representing all objects as boxed two-word buffers, using one element as the boxedness tag and another for the payload. We implemented this latter strategy, as it was slightly easier to integrate with Guile (§5.4.1).

In either case, both ε_0 primitives and the memory management system should keep boxedness tags into account: this will slow down arithmetic operations and possibly complicate garbage collection. On the other hand, some garbage collectors (for example OCaml's) already require the same tagging strategy for their own purposes: where such a collector is used anyway boxedness tags cause no additional overhead.

Since boxedness tags are expensive it is conceivable to provide two different runtime libraries, a *tagged runtime* associating a boxedness tag to every word and a length field to every buffer, and an *untagged runtime* directly using the machine representation: dumping, marshalling and unexecing will only be possible on the "tagged" runtime, but the "untagged" runtime will be more efficient. One interesting feature of this solution is that, since *unmarshalling does not rely on tags*, the untagged runtime can always be used as a last stage for a program which has been developed on the tagged runtime, before being unexeced for the final time. In case of compiled code, a (presumably static) compiled program should probably always use an untagged runtime.

Boxedness tags, when present, can also be used by primitives to perform some dynamic checks and prevent out-of-bounds errors in a very crude form of dynamic "typing", which has value when debugging. In this view it might make sense to provide *three* different runtimes: "untagged", "tagged checked" and "tagged unchecked".

Our implementation currently contains only a tagged checked runtime. Implementing the other runtimes is not hard, being mostly a matter of using C preprocessor macros to wrap object accesses; we will provide the two missing runtimes as soon as we eliminate the dependency on Guile.

[13] A memory system organized in the BiBOP style (§6) — *not necessarily a garbage collector* — would permit not to represent them at all in the most common cases.

3.3.2.2 Marshalling properties

Since we have not formally specified marshalling and unmarshalling algorithms, here we simply assert their properties without proof, as guarantees to be provided by an implementation.
Again, the strong resemblance to moving garbage collectors is not coincidental.

We first need to specify exactly what we mean as the *corresponding substructures* of an object, "before and after" marshalling:

Definition 3.5 (marshalling correspondence) *Let a_0 be an object which is marshalled into a binary dump, itself unmarshalled into the object b_0. Then, by induction:*

- *a_0 corresponds to b_0;*
- *if a corresponds to b and both a and b are pointers to n-element buffers, then the i-th component of the buffer pointed by a corresponds to the i-th component of the buffer pointed by b for any $0 \leqslant i < n$.* □

Marshalling has to "preserve structure", which is to say has to reproduce the original pointer graph, mapping buffers into buffers and unboxed objects into unboxed objects:

Axiom 3.6 *Let a correspond to b. Then we have that a is unboxed if and only if b is unboxed. For every $n \in \mathbb{N}$ we have that a is a pointer to an n-element buffer if and only if b is also a pointer to an n-element buffer.* □

Axiom 3.7 *Corresponding unboxed objects are equal provided that they both fit into a dump word payload.* □

Corresponding pointers are *not* guaranteed to be equal[14], but marshalling "preserves equality" without introducing or eliminating sharing, in the following sense:

Axiom 3.8 *Let a_1 be a pointer corresponding to b_1 and a_2 be a pointer corresponding to b_2; then we have that $a_1 = a_2$ if and only if $b_1 = b_2$.* □

As a consequence most operations over pointers continue to work with their intended semantics after unmarshalling, including checking pointer equality — but checking whether an address is numerically *smaller or bigger* then another may yield a different result.

Interpreting pointers as fixnums and doing arithmetic over them, for example to compute a hash function, in general will yield different results before and after unmarshalling. But using only the *unboxed elements* of boxed structures yields the same results after unmarshalling, except in case of overflow.

[14] We do not want to assert that they are *necessarily* different, because in practice garbage collection might intervene destroying the original object before its corresponding version is built, and it is conceivable that under unusual circumstances the unmarshalled object may reside at the same address as its corresponding original version.

3.4 Summary

Instead of hardwiring definition forms into the language syntax, we can keep the language simpler by providing procedures to update procedures and global variables. These procedures may be used anywhere, and allow for powerful self-modifying code.

"Static" code, on the other hand, has the advantage of allowing analyses and being efficiently compilable. A state where no more self-modification takes place can be reached incrementally, by successive self-modifications.

Having access to the current global state permits to save a snapshot of the system as a data structure, in a way similar to the Emacs unexec hack in terms of functionality, but implemented much more simply by data structure marshalling.

Marshalling relies on boxedness tags, which can be made optional for higher performance.

Chapter 4

A static semantics for ε_0: dimension analysis

The core language ε_0 as described in §2 is much simpler than other formally-specified languages such as SML, whose description [58, 57, 59] looks strikingly complex for a "small" language; Scheme Standards include non-normative semantics for some language subset in appendices [70, 20, 41, 79]; mainstream languages have no formal specification at all.

To make a realistic argument for the practicality of our ε_0 semantics we are going to show an example of its application by formally describing a static analysis of bundle dimensions for static programs (§3.2.2), and then proving it sound with respect to the semantics.

We chose to deal with bundle dimensions in this sample analysis because bundles are interesting as a slightly unusual feature, but of course dimension analysis has no privileged status: just like any other static analysis in ε, dimension analysis can be used as in ML for preventing runtime errors at the cost of also rejecting some correct programs, or just to obtain warnings, or not at all; and of course any number of analyses (or "type systems") can run side by side on the same program; it is up to the personality implementor to decide what to do with the results.

Contents

4.1 Dimension inference

In analogy with Hindley-Milner type inference [22] we would like to define a procedure automatically assigning[1] a *dimension* to every expression in a static program,

[1] An alternative approach based on *checking* user-supplied annotations would have been possible, since in practice only few expressions will have a dimension other than [1], which could be assumed as the default case. Inference is however even less obstrusive, and does not seem to require a substantially different formalization.

where the dimension represents a conservative approximation of the size of the bundle the expression may evaluate to at run time.

Intuitively, we want to associate dimension "one" to constants such as 42_{h_0}, and also to all variables such as x_{h_1} since non-singleton bundles are not denotable. In the same spirit, a two-object bundle such as $[\texttt{bundle}\ 10_{h_3}\ [\texttt{primitive}\ +\ 1_{h_5}\ 2_{h_6}]_{h_4}]_{h_2}$ would have dimension "two", and of course the zero-element bundle $[\texttt{bundle}\ ]_{h_7}$ would have dimension "zero".

Anyway by following this line of reasoning alone we get stuck very soon: for example, what dimension should we assign to a call to the procedure $f1$?
$[\texttt{procedure}\ (f1\ x)\ [\texttt{call}\ f2\ x_{h_2}]_{h_1}]$
$[\texttt{procedure}\ (f2\ x)\ x_{h_3}]$
$[\texttt{call}\ f1\ 42_{h_5}]_{h_4}$
Of course the answer relies on $f1$'s definition, and in particular on the dimension of its body. But $f1$'s body consists of a call to $f2$... It is already clear that dimension inference has to work *on an entire program*, using a fix point construction of some sort: in the fashion of type inference, the analysis will deduce a set of constraints from a program (for example: $f1$ returns a result with the same dimension as the result of $f2$; $f2$ returns a singleton bundle; the main expression has the same dimension as the result of $f1$), and attempt to resolve them.

4.1.1 The dimension lattice $(\mathbb{N}^{\top}_{\bot}, \sqcap, \sqcup)$

It is easy to see how our dimension domain needs to be at least slightly richer than the set of natural numbers $\mathbb{N}$, for example by looking at the main expression of the following program:
$[\texttt{procedure}\ (f\ )\ [\texttt{call}\ f\ ]_{h_1}]$
$[\texttt{call}\ f\ ]_{h_2}$
Since f never returns anything the analysis cannot discover any constraint on the dimension of its result, other than a trivial one according to which such dimension is equal to itself. We call "$\bot$" *the dimension of an expression on which we have no constraints*, such as the main expression of the program above.

As it will be made clear below, in practice only some trivially looping expressions have dimension $\bot$. From the dimension point of view such expressions are particularly unproblematic and easy to combine with others, since they can never cause failures thanks to ε_0's call-by-value strategy: for example passing a parameter with dimension $\bot$ to any unary procedure will cause an infinite loop before the body has a chance of ever being evaluated, and maybe failing.

At the opposite end of the spectrum, some expressions are clearly troublesome; for example a procedure call with a wrong number of parameters will definitely yield a dimension failure at run time, if the expression is reached and parameters converge; we assign the dimension "$\top$" to such *trivially failing* expressions.

As a slightly more subtle case, and very similarly to Hindley-Milner type inference, we need to give `if` expressions a dimension which is the "synthesis" of its branch

dimensions: when the `then` and `else` branches have incompatible dimensions, such synthesis will be $\top$. For example we assign the dimension $\top$ to the *inconsistently-dimensioned* expression $[\texttt{if } x_{h_1} \in \{1,2,3\} \texttt{ then } 10_{h_2} \texttt{ else } [\texttt{bundle }]_{h_3}]_{h_0}$; such an expression is problematic to compose, because the dimension of the result bundle varies according to which branch is taken at run time.

Our dimension domain is hence made of the natural numbers $\mathbb{N}$ extended with the two elements $\bot$ and $\top$: we call this set $\mathbb{N}_\bot^\top$. We can easily define a partial order $\_ \sqsubseteq \_$ as the reflexive closure of the relation $\_ \sqsubset \_$, where $\_ \sqsubset \_ = \bigcup_{i \in \mathbb{N}} \{(\bot, \lfloor i \rfloor), (\lfloor i \rfloor, \top)\}$.

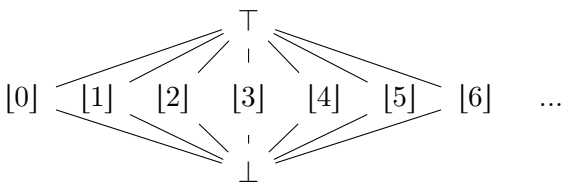

Figure 4.1: The flat lattice $(\mathbb{N}_\bot^\top, \sqcup, \sqcap)$.

The set $\mathbb{N}_\bot^\top$ with the order $\_ \sqsubseteq \_$ forms a flat lattice: for any $a, b \in \mathbb{N}_\bot^\top$, we call $a \sqcup b$ their least upper bound or "join", and $a \sqcap b$ their greatest lower bound or "meet".

In the lattice higher values correspond to more constrained dimensions, with $\bot$ representing the absence of any constraint, $\lfloor n \rfloor$ with $n \in \mathbb{N}$ representing a bundle of exactly n elements, and $\top$ expressing several conflicting constraints; the join operation $\_ \sqcup \_$ is but the "synthesis" mentioned above, yielding *the least constrained dimension which is compatible with both parameters*: joining $\bot$ with another element yields the other element, joining $\lfloor n \rfloor$ with itself yields $\lfloor n \rfloor$ for every $n \in \mathbb{N}$, and joining $\lfloor n \rfloor$ with $\lfloor m \rfloor$ for $n \neq m$ yields $\top$; joining $\top$ with any element yields $\top$.

Occasionally we may also use the set $\mathbb{N}_\bot$, defined as $\mathbb{N}_\bot^\top \backslash \{\top\}$.

4.1.2 Definition and properties

We are now ready to formally enunciate dimension analysis, computing a dimension for each expression occurring anywhere in the program, and for each procedure.

Definition 4.1 (Dimension) *Let a program p be given. We define in a mutually-recursive fashion:*

- *The dimension function for expressions, a partial function with signature $\mathbb{E} \rightharpoonup \mathbb{N}_\bot$, that we represent as the relation $\_ :_\# \_ \subseteq \mathbb{E} \times \mathbb{N}_\bot$:*

$$\frac{}{x_h :_\# \lfloor 1 \rfloor} \qquad \frac{}{c_h :_\# \lfloor 1 \rfloor}$$

$$\frac{e_{h_1} :_\# d_1 \quad \ldots \quad e_{h_n} :_\# d_n}{[\texttt{bundle}\ e_{h_1} \ldots e_{h_n}]_{h_0} :_\# \lfloor n \rfloor}\ d_i \sqsubseteq \lfloor 1 \rfloor, \text{ for all } 1 \leqslant i \leqslant n$$

$$\frac{e_{h_1} :_\# d_1 \qquad e_{h_2} :_\# d_2}{[\texttt{let}\ x_1 \ldots x_n\ \texttt{be}\ e_{h_1}\ \texttt{in}\ e_{h_2}]_{h_0} :_\# d_2}\ d_1 \sqsubseteq \lfloor m \rfloor \text{ for some } m \geqslant n,\ d_2 \sqsubset \top$$

$$\frac{\pi :_\# n \to m \qquad e_{h_1} :_\# d_1 \quad \ldots \quad e_{h_n} :_\# d_n}{[\texttt{primitive}\ \pi\ e_{h_1} \ldots e_{h_n}]_{h_0} :_\# \lfloor m \rfloor}\ d_i \sqsubseteq \lfloor 1 \rfloor, \text{ for all } 1 \leqslant i \leqslant n$$

$$\frac{f :_\# n \to d \qquad e_{h_1} :_\# d_1 \quad \ldots \quad e_{h_n} :_\# d_n}{[\texttt{call}\ f\ e_{h_1} \ldots e_{h_n}]_{h_0} :_\# d}\ d_i \sqsubseteq \lfloor 1 \rfloor, \text{ for all } 1 \leqslant i \leqslant n,\ d \sqsubset \top$$

$$\frac{e_{h_1} :_\# d_1 \qquad e_{h_2} :_\# d_2 \qquad e_{h_3} :_\# d_3}{[\texttt{if}\ e_{h_1} \in \{c_1 \ldots c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h_0} :_\# d}\ d_1 \sqsubseteq \lfloor 1 \rfloor,\ d = d_2 \sqcup d_3,\ d \sqsubset \top$$

$$\frac{f :_\# n \to d \qquad e_{h_1} :_\# d_1 \quad \ldots \quad e_{h_n} :_\# d_n}{[\texttt{fork}\ f\ e_{h_1} \ldots e_{h_n}]_{h_0} :_\# \lfloor 1 \rfloor}\ d \sqsubseteq \lfloor 1 \rfloor,\ d_i \sqsubseteq \lfloor 1 \rfloor, \text{ for all } 1 \leqslant i \leqslant n$$

$$\frac{e_{h_1} :_\# d_1}{[\texttt{join}\ e_{h_1}]_{h_0} :_\# \lfloor 1 \rfloor}\ d_1 \sqsubseteq \lfloor 1 \rfloor$$

- *the dimension function for procedures (written in relational notation)* $\_ :_\# \_ \to \_$*, with signature* $\mathbb{F} \to (\mathbb{N} \times \mathbb{N}_\bot^\top)$*, associating a procedure name with the number of its parameters and the dimension of its result.*
  For each procedure $[\texttt{procedure}\ (f\ x_1 \ldots x_n)\ e_{h_1}] \in p$ *we say* f *has in-dimension* n *and out-dimension* d*, and we write "*$f :_\# n \to d$*" where* d *is the minimum fixpoint such that* $\#(e_{h_1}) = d$*.*

Then we define $\#(\_) : \mathbb{E} \to \mathbb{N}_\bot^\top$ *as the total extension of* $\_ :_\# \_$*, so that* $\#(\_)$ *returns* $\top$ *where* $\_ :_\# \_$ *is not defined and its same result elsewhere.* □

Definition 4.1 depends on the fact that the relation $\_ :_\# \_$ be a function, which is clearly true because rule premises are pairwise disjoint.

Notice that all the side constraints of the form "$d \sqsubset \top$" (rules for `let`, `call` and `if`) are only included for aesthetic symmetry, so that in case of any dimension inconsistency $\_ :_\# \_$ remains undefined just as in the other syntactic cases, rather than returning $\top$: of course the total extension $\#(\_)$ would remain the same even if we erased such side constraints from $\_ :_\# \_$.

Definition 4.2 *We call an expression* e_h *plural if* $\#(e_h) = \lfloor n \rfloor$ *for some* $n \neq 1$*, consistently-dimensioned if* $\#(e_h) \sqsubset \top$ *and inconsistently-dimensioned if* $\#(e_h) = \top$*.* □

We intentionally wrote Definition 4.2 so that it makes empty bundles plural, and trivially-looping expressions not plural. The reason for this choice is bound to the implementation: only what we call plural expressions requires some non-conventional implementation technique such as placing a value in a number of registers or stack slots different from one. Expressions which never return anything do not pose particular problems — and we stress again that we do *not* consider non-termination an error; anyway the existence of expressions $e_h :_\# \bot$ is the reason why we resist the temptation of defining "singular" expressions; should e_h be both singular and plural, singular and not plural, plural and not singular, or neither? No solution seems intuitive, or particularly useful.

It is not hard to see how inconsistent dimensioning "propagates outwards", from a *contained* expression out to its *containing* expression:

Proposition 4.3 *Any expression containing an inconsistently-dimensioned subexpression is inconsistently-dimensioned itself.*

Proof Assuming $e_h :_\# \top$, we have to prove that $C[e_h] :_\# \top$ for all contexts $C[\_]$. A straightforward structural induction over contexts:

- $C[e_h] = e_h$ (base case): we trivially have that $\#(C[e_h]) = \#(e_h) = \top$;
- $C[e_h] = [\texttt{let}\ x_1 ... x_n\ \texttt{be}\ C'[e_h]\ \texttt{in}\ e_{h_2}]_{h_0}$: since by hypothesis $\#(e_h) = \top$, by induction hypothesis we also have $\#(C'[e_h]) = \top \not\equiv \lfloor n \rfloor$; this makes impossible to satisfy the conditions of the `let` rule in Definition 4.1; so the relation $\_ :_\# \_$ is undefined on $C[e_h]$, and again by Definition 4.1 we have that $\#(C[e_h]) = \top$;
- $C[e_h] = [\texttt{let}\ x_1 ... x_n\ \texttt{be}\ e_{h_1}\ \texttt{in}\ C'[e_h]]_{h_0}$: again we have that $\#(C'[e_h]) = \top$, and the `let` rule in Definition 4.1 cannot fire because the `let` body $C'[e_h]$ has dimension $\top$; if the rule does not fire then $C[e_h] = \top$ because $\_ :_\# \_$ is undefined on the parameter, as in the previous case;
- $C[e_h] = [\texttt{call}\ f\ e_{h_1} ... e_{h_n} C'[e_h] e_{h_{n+1}} ... e_{h_{n+m}}]_{h_0}$: again $\#(C'[e_h]) = \top$ by induction hypothesis; but then there exist a procedure actual whose dimension is not lower than or equal to $\lfloor 1 \rfloor$, and the `call` rule in Definition 4.1 cannot fire; $\_ :_\# \_$ is undefined on $C[e_h]$, hence $\#(C[e_h]) = \top$;
- $C[e_h] = [\texttt{primitive}\ \pi\ e_{h_1} ... e_{h_n} C'[e_h] e_{h_{n+1}} ... e_{h_{n+m}}]_{h_0}$: same reasoning as the `call` case;
- $C[e_h] = [\texttt{if}\ C'[e_h] \in \{c_1 ... c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h_0}$: again $\#(C'[e_h]) = \top$ by induction hypothesis, which means that the dimension of the discriminand is not lower than or equal to $\lfloor 1 \rfloor$, the `if` rule in Definition 4.1 cannot fire, hence $\_ :_\# \_$ is undefined on $C[e_h]$, and $\#(C[e_h]) = \top$;
- $C[e_h] = [\texttt{if}\ e_{h_1} \in \{c_1 ... c_n\}\ \texttt{then}\ C'[e_h]\ \texttt{else}\ e_{h_3}]_{h_0}$: similar to the `let` body case: $\#(C'[e_h]) = \top$, which prevents the `if` rule in Definition 4.1 from firing;

- $C[e_h] = [\texttt{if } e_{h_1} \in \{c_1 ... c_n\} \texttt{ then } e_{h_2} \texttt{ else } C'[e_h]]_{h_0}$: same reasoning as the previous case;
- $C[e_h] = [\texttt{fork } f \ e_{h_1} ... e_{h_n} C'[e_h] e_{h_{n+1}} ... e_{h_{n+m}}]_{h_0}$: same reasoning as the `call` case;
- $C[e_h] = [\texttt{join } C'[e_h]]_{h_0}$: same reasoning as the `call` case;
- $C[e_h] = [\texttt{bundle } e_{h_1} ... e_{h_n} C'[e_h] e_{h_{n+1}} ... e_{h_{n+m}}]_{h_0}$: same reasoning as the `call` case. ■

It is also intuitive that replacing a subexpression with another whose dimension is *lower* or equal will not raise the dimension of the containing expression, which makes $\#(\_)$ a monotonic function:

Proposition 4.4 (#-monotonicity) *Replacing a subexpression with another of lower or equal dimension cannot raise the dimension of the containing expression. More formally, for any expression context $C[\_]$, expression e and expression e', if we have that $\#(e) \sqsubseteq \#(e')$ then we also have that $\#(C[e]) \sqsubseteq \#(C[e'])$.*

Proof Another straightforward structural induction over contexts. ■

The following definition identifies programs where no expression of dimension $\top$ occurs anywhere. As the reader will have anticipated, we are going to prove the condition *sufficient* to guarantee a desirable property with respect to the dynamic semantics.

Definition 4.5 (well-dimensioned) *We call the static program p well-dimensioned if both the following conditions hold:*

- *for all procedure definitions* $[\texttt{procedure (}f\ x_1 ... x_n\texttt{)}\ e_{h_1}] \in p$ *such that* $f :_{\#} n \to d$, *we have* $d \sqsubset \top$;
- *for the main expression* $e_{h_2} \in p$ *we have that* $\#(e_{h_2}) \sqsubset \top$.

We call ill-dimensioned all programs which are not well-dimensioned. □

4.1.2.1 There cannot be a *most precise* dimension analysis

It would be nice to be able to characterize our definition of an expression dimension as the "most precise", but unfortunately our definition is not the best one, and in fact no such definition can exist.

In order to see why at least at an intuitive level we consider the program:

$[\texttt{procedure (}loop\texttt{)}\ [\texttt{call } loop\ ]_{h_1}] \in p$
$[\texttt{if } \mathcal{N}(1)_{h_3} \in \{\mathcal{N}(2)\} \texttt{ then } \mathcal{N}(42)_{h_4} \texttt{ else } [\texttt{call } loop\ ]_{h_5}]_{h_2} \in p$

It is obvious that the main expression loops, but the analysis assigns the main

expression the dimension $\lfloor 1 \rfloor$ instead of $\bot$: hence our definition of dimension does not correspond to "the best possible" analysis, as it is possible to change it to account for more particular cases, yielding a more precise result: in fact we can always improve the analysis by recognizing particular patterns in programs — for example by simplifying statically-determined conditionals at compile time as a first refinement; but because of the Halting Theorem we cannot hope to cover *all* possible cases.

This trivial fact prevents us from finding a result similar to the Most-General-Type theorem in [22].

4.2 Semantic soundness

Before proving the result connecting dimension analysis with ε_0's dynamic semantics we need to define some machinery.

4.2.1 Resynthesization

The idea of *resynthesization* consists in taking any reachable configuration χ and reconstructing from it an expression e that, if evaluated at the top level in the state of χ, would yield the same result and the same effects as χ. Actually we do not need to specify this equivalence any further, and in fact we will not prove any result such as reduction-equivalence on resynthesization, since our use of it here is very well-delimited, due to the technical need of assigning a dimension to all reachable ε_0 configurations.

We can easily view the content of any value stack V in a reachable configuration as a *list of non-holed expressions* E_V, by remarking the intuitive role of "$\wr$" as a bundle delimiter; bundles within V can be represented in E_V as explicit `bundle` expressions[2]. For the purposes of resynthesization it is also safe to ignore "‡" delimiters, since the particular arity mismatches they were conceived to prevent (see §2.5.1) cannot occur in static programs, our only programs of interest in this chapter.

More formally, we define the translation as follows:

$E_{\wr} \triangleq \langle\rangle$
$E_{\wr \ddagger V} \triangleq E_{\wr V}$,
$E_{\wr c_1 \ldots c_n \wr V} \triangleq [\texttt{bundle}\ c_1 \ldots c_n]_{h'}.E_{\wr V}$ with some fresh handle h'.

For example $V = \wr 1\ 2 \wr 3 \wr$ would be transformed into $E_V = \langle [\texttt{bundle}\ 1_{h'_1} 2_{h'_2}]_{h'_0}, [\texttt{bundle}\ 3_{h'_4}]_{h'_3} \rangle$, with fresh handles h'_0, h'_1, h'_2, h'_3 and h'_4.

We define resynthesization as a relation r, written in functional notation as $r(\_\ \_)$;

[2] Actually we would need to introduce explicit `bundle` expressions only for *plural* bundles in V; the definition given below avoids this complication at the price of producing some trivial `bundle` expressions with only one item.

for readability's sake we omit the comma between the two parameters of r, since both tend to be syntactically complex.

Given *a stack* and *a list of non-holed expressions* as obtained from the translation above, resynthesization produces *a non-holed expression list*:

Definition 4.6 (resynthesization) *We define the resynthesization relation $r(\_\ \_)$ as follows, with the convention that all the prime-decorated handles only appearing on the right sides be fresh:*

- $r(\langle\rangle\ E) = E$
- $r((e_h,\ \rho).S\ E) = r(S\ e_h.E)$, *for any non-holed* e_h*;*
- $r(([\texttt{let}\ x_1...x_n\ \texttt{be}\ \square\ \texttt{in}\ e_{h_2}]_{h_0},\ \rho).S\ e_a.E)$
  $= r(([\texttt{let}\ x_1...x_n\ \texttt{be}\ e_a\ \texttt{in}\ e_{h_2}]_{h'_0},\ \rho).S\ E)$
- $r(([\texttt{call}\ f\ \square]_{h_0},\ \rho).S\ e_{a_n}e_{a_{n-1}}...e_{a_2}e_{a_1}.E)$
  $= r(([\texttt{call}\ f\ e_{a_1}...e_{a_n}]_{h'_0},\ \rho).S\ E)$
- $r(([\texttt{primitive}\ \pi\ \square]_{h_0},\ \rho).S\ e_{a_n}e_{a_{n-1}}...e_{a_2}e_{a_1}.E)$
  $= r(([\texttt{primitive}\ \pi\ e_{a_1}...e_{a_n}]_{h'_0},\ \rho).S\ E)$
- $r(([\texttt{if}\ \square \in \{c_1...c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h_0},\ \rho).S\ e_a.E)$
  $= r(([\texttt{if}\ e_a \in \{c_1...c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h'_0},\ \rho).S\ E)$
- $r(([\texttt{bundle}\ \square]_{h_0},\ \rho).S\ e_{a_n}e_{a_{n-1}}...e_{a_2}e_{a_1}.E)$
  $= r(([\texttt{bundle}\ e_{a_1}...e_{a_n}]_{h'_0},\ \rho).S\ E)$
- $r(([\texttt{fork}\ f\ \square]_{h_0},\ \rho).S\ e_{a_n}e_{a_{n-1}}...e_{a_2}e_{a_1}.E)$
  $= r(([\texttt{fork}\ f\ e_{a_1}...e_{a_n}]_{h'_0},\ \rho).S\ E)$
- $r(([\texttt{join}\ \square]_{h_0},\ \rho).S\ e_a.E)$
  $= r(([\texttt{join}\ e_a]_{h'_0},\ \rho).S\ E)$ □

It is obvious from Definition 4.6 that resynthesization is deterministic up to handle identity, and the same can be said about the value stack conversion defined above. Since the specific choice of handles is immaterial with respect to dimension, and accounting for the specific choice of handles would make resynthesization much harder to work with without any particular benefit, from now on we will commit a slight abuse of language and speak about resynthesization *as a function.*

In the following we are also going to need a couple of simple properties of resynthesization:

Lemma 4.7 ("r does not delete expressions") *If $r(S\ E) = E'$ for some S, E and E', then all expressions occurring in E also occur somewhere in E'.*
More formally, for all e, if $r(S\ E_1.C[e].E_2) = E'$, then there exist $E'_1, C'[\_], E'_2$ such that $E' = E'_1.C'[e].E'_2$.

PROOF By induction on the number of recursive calls to r. ∎

Lemma 4.8 (resynthesization shape-independence) *The shape of the expressions contained in E does not affect the result of $r(S\ E)$.*
More formally, for all expression sequences E_1, E_2, E'_1, E'_2, contexts $C[\_], C'[\_]$ and expression e, if we have that $r(S\ E_1.C[e].E_2) = E'_1.C'[e].E'_2$ then we also have that $r(S\ E_1.C[e'].E_2) = E'_1.C'[e'].E'_2$ for any other expression e'.

PROOF By induction on the number of recursive calls to r. ∎

4.2.2 Weak dimension preservation

Resynthesization allows us to gloss over the difference between an ε_0 *expression* and any *configuration* reached by evaluating an ε_0 expression, so that we may speak about the dimension of either; so, in order to further simplify our presentation, we extend Definition 4.1 by assigning a dimension also to reachable configurations: a reachable configuration will have the dimension of its resynthesization; or, slightly more formally, if $\chi = (S\ V\ \Gamma)$ is a reachable configuration, then we write "$\#(\chi)$" to mean "$\#(e)$, where $r(S\ E_V) = \langle e \rangle$".

It is not yet clear at this point why r is always defined and always yields a singleton expression sequence on reachable configurations; we defer the proof to Corollary 4.10.

The *Weak Dimension Preservation* property, below, is the central result bridging ε_0's dynamic semantics to dimension analysis by showing that "evaluation preserves dimension"; in some circles such properties are known as "subject reductions" — or more intuitively in French as *auto-réductions*.

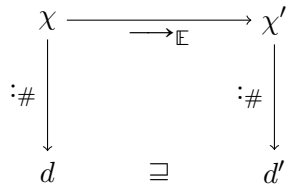

Figure 4.2: The *Weak Dimension Preservation property* (Lemma 4.9): when a configuration χ of dimension d can reduce to another configuration χ' of dimension d' we have that $d' \sqsubseteq d$.

The property is "weak" in the sense that an expression is allowed to reduce to another expression of *lower* dimension: as it can be seen from the proof, this may happen with conditionals: choosing one branch or the other entails replacing the expression on the top of the stack by a subexpression of it, hence by an expression with fewer dimension constraints. In particular it is possible that an inconsistently-dimensioned expression reduces to a consistently-dimensioned one ("from $\top$ to $d \sqsubset \top$"); anyway the vice-versa ("from $d \sqsubset \top$ to $\top$") *cannot* happen, which is all we need for our soundness property.

Lemma 4.9 (Weak Dimension Preservation) *Let reachable configurations χ, χ' be given, such that $\chi = (S_\chi\ V_\chi\ \Gamma_\chi) \longrightarrow_{\mathbb{E}} \chi' = (S_{\chi'}\ V_{\chi'}\ \Gamma_{\chi'})$; now, if $r(S_\chi\ E_{V_\chi}) = \langle e \rangle$ then there exists e' such that $r(S_{\chi'}\ E_{V_{\chi'}}) = \langle e' \rangle$ and $\#(e') \sqsubseteq \#(e)$.*

Proof Induction is not needed: we just directly prove that, for all cases in which $(S_\chi\ V_\chi\ \Gamma_\chi) = \chi \longrightarrow_{\mathbb{E}} \chi' = (S_{\chi'}\ V_{\chi'}\ \Gamma_{\chi'})$, we have that $\#(r(S_{\chi'}\ E_{V_{\chi'}})) \sqsubseteq \#(r(S_\chi\ E_{V_\chi}))$. We avoid writing handles for converted value stack expressions, since they are immaterial anyway; in this proof we also freely abuse the notation by saying that r returns results which are "equal to" something else, writing "=" without explicitly stating that the equality is up to handle choice.

- [*constant*]
  $(c_h,\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} S\ \wr c \wr V\ \Gamma$:
  $r(S_\chi\ E_{V_\chi})$ = {definition of r} $r(S\ c.E_{V_\chi})$ = {substitution} $r(S_{\chi'}\ E_{V_{\chi'}})$; hence in this case we have that $\#(r(S_{\chi'}\ E_{V_{\chi'}})) = \#(r(S_\chi\ E_{V_\chi}))$;

- [*variable*]
  $(x_h,\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} S\ \wr c \wr V\ \Gamma$:
  looking at χ, we have $r(S_\chi\ E_{V_\chi})$ = {by definition of r} $r(S\ x.E_{V_\chi})$ = {hypothesis} $\langle e \rangle$ = {Lemma 4.7, for some $C[\_]$} $\langle C[x] \rangle$;
  looking at χ', we have $r(S_{\chi'}\ E_{V_{\chi'}})$ = {substitution} $r(S\ c.E_{V_\chi})$ = {Lemma 4.8} $\langle C[c] \rangle$, which we call $\langle e' \rangle$; since $\#(c) = \lfloor 1 \rfloor \sqsubseteq \#(x) = \lfloor 1 \rfloor$, by Proposition 4.4 we have that $\#(e') \sqsubseteq \#(e)$;

- [$\texttt{let}_e$]
  $([\texttt{let}\ x_1 ... x_n\ \texttt{be}\ e_{h_1}\ \texttt{in}\ e_{h_2}]_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho).([\texttt{let}\ x_1 ... x_n\ \texttt{be}\ \square\ \texttt{in}\ e_{h_2}]_{h_0},\ \rho).S\ \wr V\ \Gamma$:
  looking at χ we have that $r(S_\chi\ E_{V_\chi})$ = {definition of r}
  $r(S\ [\texttt{let}\ x_1 ... x_n\ \texttt{be}\ e_{h_1}\ \texttt{in}\ e_{h_2}]_{h_0}.E_{V_\chi})$; looking at χ' we have that $r(S_{\chi'}\ E_{V_{\chi'}})$ = {definition of r} $r(([\texttt{let}\ x_1 ... x_n\ \texttt{be}\ \square\ \texttt{in}\ e_{h_2}]_{h_0}, \rho).S\ e_{h_1}.E_{V_\chi})$ = {definition of r} $r(([\texttt{let}\ x_1 ... x_n\ \texttt{be}\ e_{h_1}\ \texttt{in}\ e_{h_2}]_{h'_0}, \rho).S\ E_{V_\chi})$ = {substitution}
  $r(S\ [\texttt{let}\ x_1 ... x_n\ \texttt{be}\ e_{h_1}\ \texttt{in}\ e_{h_2}]_{h'_0}.E_{V_\chi})$ = {substitution} $r(S_\chi\ E_{V_\chi})$ = {hypothesis} $\langle e \rangle$: again we have that $r(S_\chi\ E_{V_\chi})$ and $r(S_{\chi'}\ E_{V_{\chi'}})$ are equal to the same singleton sequence $\langle e \rangle = \langle e' \rangle$, hence $\#(e') = \#(e)$; this proof case is essentially identical to the proof cases of the other expansive rules;

- [$\texttt{let}_c$]
  $([\texttt{let}\ x_1 ... x_n\ \texttt{be}\ \square\ \texttt{in}\ e_{h_2}]_{h_0},\ \rho).S\ \wr c_m c_{m-1} ... c_2 c_1 \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_2},\ \rho[x_1 \mapsto c_1, x_2 \mapsto c_2, ..., x_n \mapsto c_n]).S\ \wr V\ \Gamma$:
  looking at χ we find that $r(S_\chi\ E_{V_\chi})$ = {definition of r, twice}
  $r(S\ [\texttt{let}\ x_1 ... x_n\ \texttt{be}\ [\texttt{bundle}\ c_1 ... c_m]_{h'_1}\ \texttt{in}\ e_{h_2}]_{h'_0}.E_{V_{\chi'}})$ = {hypothesis} $\langle e \rangle$ = {Lemma 4.7, for some context $C[\_]$} $\langle C[[\texttt{let}\ x_1 ... x_n\ \texttt{be}\ [\texttt{bundle}\ c_1 ... c_m]_{h'_1}\ \texttt{in}\ e_{h_2}]_{h'_0}] \rangle$;
  looking at χ' we have that $r(S_{\chi'}\ E_{V_{\chi'}})$ = {definition of r, twice} $r(S\ e_{h_2}.E_{V_{\chi'}})$ = {Lemma 4.8} $\langle C[e_{h_2}] \rangle$, which we call $\langle e' \rangle$; since by Definition 4.1 a `let` expression has the same dimension as its body, it follows that $\#(e') \sqsubseteq \#(e)$ by Proposition 4.4;

- $[\texttt{call}_e]$
  $(\llbracket \texttt{call}\ f\ e_{h_1}...e_{h_n} \rrbracket_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho)...(e_{h_n},\ \rho).(\llbracket \texttt{call}\ f\ \Box \rrbracket_{h_0},\ \varnothing).S\ \wr\ddagger V\ \Gamma$:
  identical to the other expansive rule cases;

- $[\texttt{call}_c]$
  $(\llbracket \texttt{call}\ f\ \Box \rrbracket_{h_0},\ \rho).S\ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr\ddagger V\ \Gamma \longrightarrow_{\mathbb{E}} (e_h,\ \rho[x_1 \mapsto c_1, x_2 \mapsto c_2, ..., x_{n-1} \mapsto c_{n-1}, x_n \mapsto c_n]).S\ \wr V\ \Gamma$:
  by the rule side condition f takes exactly n parameters and has dimension $n \to d$ for some d. Looking at χ we find that $r(S_\chi\ E_{V_\chi}) =$ {definition of r} $r((\llbracket \texttt{call}\ f\ c_1...c_n \rrbracket_{h'_0}, \rho).S\ E_{V_{\chi'}}) =$ {hypothesis and Lemma 4.7, for some context $C[\_]$} $\langle C[\llbracket \texttt{call}\ f\ c_1...c_n \rrbracket_{h'_0}] \rangle$.
  Starting at χ', $r(S_{\chi'}\ E_{V_{\chi'}}) =$ {substitution} $r((e_h, \rho).S\ E_{V_{\chi'}}) =$ {Lemma 4.8} $\langle C[e_h] \rangle$. But e_h and $\llbracket \texttt{call}\ f\ c_1...c_n \rrbracket_{h'_0}$ have the same dimension d by Definition 4.1, hence by Proposition 4.4 we have that $\#(C[e_h]) \sqsubseteq \#(C[\llbracket \texttt{call}\ f\ c_1...c_n \rrbracket_{h'_0}])$;

- $[\texttt{primitive}_e]$
  $(\llbracket \texttt{primitive}\ \pi\ e_{h_1}...e_{h_n} \rrbracket_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}}$
  $(e_{h_1},\ \rho)...(e_{h_n},\ \rho).(\llbracket \texttt{primitive}\ \pi\ \Box \rrbracket_{h_0},\ \varnothing).S\ \wr\ddagger V\ \Gamma$:
  identical to the other expansive rule cases;

- $[\texttt{primitive}_c]$
  $(\llbracket \texttt{primitive}\ \pi\ \Box \rrbracket_{h_0},\ \rho).S\ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr\ddagger V\ \Gamma \longrightarrow_{\mathbb{E}} S\ \wr c'_m c'_{m-1} ... c'_2 c'_1 \wr V\ \Gamma'$, when $\Gamma_{\texttt{primitives}}(\pi)(c_1, ..., c_n, \Gamma) = \langle c'_1, ..., c'_m, \Gamma' \rangle$:
  since the rule side condition applies, we have that $\pi :_{\#} n \to m$;
  $r(S_\chi\ E_{V_\chi}) =$ {definition, twice} $r(S\ \langle \llbracket \texttt{primitive}\ \pi\ c_1...c_n \rrbracket_{h'_0} \rangle . E_{V_\chi}) =$ {hypothesis, for some context $C[\_]$} $\langle C[\llbracket \texttt{primitive}\ \pi\ c_1...c_n \rrbracket_{h'_0}] \rangle$;
  $r(S_{\chi'}\ E_{V_{\chi'}}) =$ {substitution} $r(S\ \llbracket \texttt{bundle}\ c'_1...c'_m \rrbracket_{h'} . E_{V_\chi}) =$ {Lemma 4.8} $\langle C[\llbracket \texttt{bundle}\ c'_1...c'_m \rrbracket_{h'}] \rangle$; since by Definition 4.1 $\llbracket \texttt{bundle}\ c'_1...c'_m \rrbracket_{h'}$ and $\llbracket \texttt{primitive}\ \pi\ c_1...c_n \rrbracket_{h'_0}$ have the same dimension, we conclude by Proposition 4.4;

- $[\texttt{if}_e]$
  $(\llbracket \texttt{if}\ e_{h_1} \in \{c_1...c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3} \rrbracket_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho).(\llbracket \texttt{if}\ \Box \in \{c_1...c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3} \rrbracket_{h_0},\ \rho).S\ \wr V\ \Gamma$:
  identical to the other expansive rule cases;

- $[\texttt{if}^{\in}_c]$
  $(\llbracket \texttt{if}\ \Box \in \{c_1...c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3} \rrbracket_{h_0},\ \rho).S\ \wr c \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_2},\ \rho).S\ \wr V\ \Gamma$, when $c \in \{c_1...c_n\}$:
  $r(S_\chi\ E_{V_\chi}) =$ {definition, twice} $r(S\ \llbracket \texttt{if}\ c \in \{c_1...c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3} \rrbracket_{h'_0} . E_{V_\chi}) =$ {hypothesis and Lemma 4.7, for some context $C[\_]$}
  $\langle C[\llbracket \texttt{if}\ c \in \{c_1...c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3} \rrbracket_{h'_0}] \rangle$;
  $r(S_{\chi'}\ E_{V_{\chi'}}) =$ {definition} $r(S\ e_{h_2} . E_{V_\chi}) =$ {Lemma 4.8} $\langle C[e_{h_2}] \rangle$;

by Definition 4.1 we have that $\#(e_{h_2}) \sqsubseteq \#([\texttt{if } c \in \{c_1...c_n\} \texttt{ then } e_{h_2} \texttt{ else } e_{h_3}]_{h'_0})$, and we conclude by Proposition 4.4;

- $[\texttt{if}_c^{\notin}]$
  $([\texttt{if } \square \in \{c_1...c_n\} \texttt{ then } e_{h_2} \texttt{ else } e_{h_3}]_{h_0},\ \rho).S\ \wr c \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_3},\ \rho).S\ \wr V\ \Gamma$, when $c \notin \{c_1...c_n\}$:
  identical to the previous case;

- $[\texttt{bundle}_e]$
  $([\texttt{bundle } e_{h_1}...e_{h_n}]_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho)...(e_{h_n},\ \rho).([\texttt{bundle } \square]_{h_0},\ \varnothing).S\ \wr\ddagger V\ \Gamma$:
  identical to the other expansive rule cases;

- $[\texttt{bundle}_c]$
  $([\texttt{bundle } \square]_{h_0},\ \rho).S\ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr\ddagger V\ \Gamma \longrightarrow_{\mathbb{E}} S\ \wr c_n c_{n-1}...c_2 c_1 \wr V\ \Gamma$:
  $r(S_\chi\ E_{V_\chi}) =$ {definition, twice} $r(S\ [\texttt{bundle } c_1...c_n]_{h'}.E) =$ {substitution} $r(S_{\chi'}\ E_{V_{\chi'}})$, and again we have that $e = e'$;

- $[\texttt{fork}_e]$
  $([\texttt{fork } f\ e_{h_1}...e_{h_n}]_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho)...(e_{h_n},\ \rho).([\texttt{fork } f\ \square]_{h_0},\ \varnothing).S\ \wr\ddagger V\ \Gamma$:
  identical to the other expansive rule cases;

- $[\texttt{fork}_c]$
  $([\texttt{fork } f\ \square]_{h_0},\ \rho).S\ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr\ddagger V\ \Gamma \longrightarrow_{\mathbb{E}} S\ \wr \mathcal{T}(t) \wr V\ \Gamma[^{t \mapsto ((e_h,\ \rho[x_0 \mapsto \mathcal{T}(t), x_1 \mapsto c_1, ..., x_n \mapsto c_n])\ \wr)}_{\texttt{futures}}]$:
  $r(S_\chi\ E_{V_\chi}) =$ {definition, twice} $r(S\ [\texttt{fork } f\ c_1...c_n]_{h'_0}.E) =$ {hypothesis, for some context $C[\_]$} $\langle C[[\texttt{fork } f\ c_1...c_n]_{h'_0}]\rangle$;
  $r(S_{\chi'}\ E_{V_{\chi'}}) =$ {definition} $r(S\ \mathcal{T}(t).E) =$ {Lemma 4.8} $\langle C[\mathcal{T}(t)]\rangle$. Since by Definition 4.1 we have that $\#(\mathcal{T}(t)) = \lfloor 1 \rfloor \sqsubseteq \#([\texttt{fork } f\ c_1...c_n]_{h'_0})$, we conclude with Proposition 4.4;

- $[\texttt{join}_e]$
  $([\texttt{join } e_{h_1}]_{h_0},\ \rho).S\ \wr V\ \Gamma \longrightarrow_{\mathbb{E}} (e_{h_1},\ \rho).([\texttt{join } \square]_{h_0},\ \rho).S\ \wr V\ \Gamma$:
  identical to the other expansive rule cases;

- $[\texttt{join}_c]$
  $([\texttt{join } \square]_{h_0},\ \rho).S\ \wr \mathcal{T}(t) \wr V\ \Gamma \longrightarrow_{\mathbb{E}} S\ \wr c_t \wr V\ \Gamma$, when $\Gamma_{\texttt{futures}} : t \mapsto (\langle\rangle,\ \wr c_t)$:
  $r(S_\chi\ E_{V_\chi}) =$ {definition, twice} $r(S\ [\texttt{join } \mathcal{T}(t)]_{h'_0}.E) =$ {hypothesis and Lemma 4.7, for some context $C[\_]$} $\langle C[[\texttt{join } \mathcal{T}(t)]_{h'_0}]\rangle$;
  $r(S_{\chi'}\ E_{V_{\chi'}}) =$ {substitution} $r(S\ c_t.E) =$ {Lemma 4.8} $\langle C[c_t]\rangle$;
  since by Definition 4.1 we have that $\#(c_t) = \lfloor 1 \rfloor \sqsubseteq \#([\texttt{join } \mathcal{T}(t)]_{h'_0})$, we conclude by Proposition 4.4;

- $[\|]$
  $S\ V\ \Gamma \longrightarrow_{\mathbb{E}} S\ V\ \Gamma'[^{t \mapsto (S'_t,\ V'_t)}_{\texttt{futures}}]$, when $\Gamma_{\texttt{futures}} : t \mapsto (S_t,\ V_t)$ and $S_t\ V_t\ \Gamma \longrightarrow_{\mathbb{E}} S'_t\ V'_t\ \Gamma'$:
  here $r(S_\chi\ E_{V_\chi}) = r(S_{\chi'}\ E_{V_{\chi'}})$, hence e' exists and is equal to e. ■

The following trivial consequence of Lemma 4.9 allows us to think of r as always returning *a single* non-holed expression, when applied on a stack and a value stack (re-encoded as a list of non-holed expressions) from a reachable configuration:

Corollary 4.10 *Reachable configurations resynthesize into exactly one expression.*

PROOF The property is obvious for initial configurations, which make up the induction base; Lemma 4.9 proves the inductive case. ∎

4.2.3 Semantic soundness properties

In the style of the *Semantic Soundness* Theorem of [55, §3.7], we can now finally prove that "well-dimensioned programs do not go wrong":

Theorem 4.11 (Dimension Semantic Soundness) *No consistently-dimensioned expression fails because of dimension: more formally, for all e_h and Γ, if $\#(e_h) \sqsubset \top$ then for each χ such that $((e_h, \varnothing) \wr \Gamma) \longrightarrow_{\mathbb{E}}^{*} \chi$ we cannot have that $\chi \longrightarrow_{\mathbb{E}} ※_{\#}$.*

PROOF By contradiction, let us assume that a reachable consistently-dimensioned expression e_h fails because of dimension: then we have that $\#(e_h) \sqsubset \top$ and $((e_h, \varnothing) \wr \Gamma) \longrightarrow_{\mathbb{E}}^{*} \chi \longrightarrow_{\mathbb{E}} ※_{\#}$; but because of Lemma 4.9 each reduction starting from the initial configuration either leaves the dimension unchanged or lowers it, hence $\#(\chi) \sqsubseteq \#((e_h,\ \varnothing)\ \langle\rangle) \sqsubset \top$, which means that $r(\chi)$ is also consistently-dimensioned.
We examine all the possible cases where $\chi \longrightarrow_{\mathbb{E}} ※_{\#}$:

- $([\texttt{let}\ x_1 ... x_n\ \texttt{be}\ \square\ \texttt{in}\ e_{h_2}]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}$ when the top bundle on V has less than n elements, let us say $c'_1 ... c'_k$ with $k < n$: then by Definition 4.6 applied twice and Lemma 4.7 for some context $C[\_]$ we have that $r(\chi) = C[[\texttt{let}\ x_1 ... x_n\ \texttt{be}\ c'_1 ... c'_k\ \texttt{in}\ e_{h_2}]_{h_0}]$; but then by Definition 4.1 the `let` expression cannot be consistently-dimensioned, and neither can $r(\chi)$ by Proposition 4.3: contradiction;

- $([\texttt{call}\ f\ \square]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}$ when the top frame on the value stack has a wrong number of $\wr$-separated constants: then by Definition 4.6 and Definition 4.1 we have that $r(\chi) = \top$: contradiction;

- $([\texttt{primitive}\ \pi\ \square]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}$ when $\pi :_{\#} n \to m$, $V \not\equiv\ \wr c_n \wr c_{n-1} \wr ... \wr c_2 \wr c_1 \wr \ddagger V'$:
  identical to the $[\texttt{call}\ f\ \square]_{h_0}$ case;

- $([\texttt{if}\ \square \in \{c_1 ... c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}$ when $V \not\equiv\ \wr c \wr V'$:
  by Definition 4.6 and Definition 4.1 we find immediately that $r(\chi) = \top$: contradiction;

- $([\texttt{bundle}\ \square]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}$ when $V \not\equiv\ \wr c_1 \wr c_2 \wr ... \wr c_{n-1} \wr c_n \wr V'$:
  similar to the $[\texttt{call}\ f\ \square]_{h_0}$ case: since χ is reachable the original `bundle` expression contained exactly n parameters, but some of them are plural: however by Definition 4.6 and Definition 4.1 we have $r(\chi) = \top$: contradiction;

- $([\texttt{fork}\ f\ \Box]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}$ when the top frame on the value stack has a wrong number of $\wr$-separated constants: identical to the $[\texttt{call}\ f\ \Box]_{h_0}$ case;

- $([\texttt{join}\ \Box]_{h_0},\ \rho).S\ V\ \Gamma \longrightarrow_{\mathbb{E}} ※_{\#}$ when $V \not\equiv\ \wr c \wr V'$:
identical to the $[\texttt{if}\ \Box \in \{c_1 ... c_n\}\ \texttt{then}\ e_{h_2}\ \texttt{else}\ e_{h_3}]_{h_0}$ case. ■

Corollary 4.12, a simple consequence of Theorem 4.11, extends the semantic soundness result to whole programs by providing a sufficient condition for avoiding dimension errors.

Corollary 4.12 *Well-dimensioned-programs cannot fail because of dimension.* ▫

Of course well-dimensioning is only a sufficient condition for the absence of dimension failures: an expression containing unreachable code such as $[\texttt{if}\ \mathcal{N}(1)_{h_1} \in \{\mathcal{N}(2)\}\ \texttt{then}\ [\texttt{bundle}\ 3_{h_3}\ 4_{h_4}]_{h_2}\ \texttt{else}\ 5_{h_5}]_{h_0}$ may have dimension $\top$, without ever failing because of dimension.

4.3 Reminder: why we accept ill-dimensioned programs

Even when only speaking about static programs, we prefer **not** to restrict ourselves to well-dimensioned programs for reasons of philosophical coherency (§1.4), despite the difficulty of finding believable examples of ill-dimensioned programs that we would like to accept as "correct".

We could argue that it is at least conceivable that syntactic extensions automatically produce static but ill-dimensioned ε_0 programs — which maybe could be proved not to fail because of dimension, due to some property enjoyed by the extension. On the other hand the extension might actually be unsafe in the general case, and still useful.

But independently of any extension, at the level of the core language it is reasonable to accept any program which *could* possibly yield a useful output: since we want to respect the programmers' intelligence ε_0 will not constrain the expressivity of the upper layers; therefore we want to accept ill-dimensioned programs *and run them* until an error condition is reached, if ever. The compiler should generate code which runs until possible, and compilation itself should **not** fail because of ill-dimensioning.

Of course a personality implementer is always free to add static checks generating warning messages or even fatal errors at compile time, yielding a very safe — if restrictive — language. Such languages do have a place in the world, as shown by the experience of Ada, ML and Haskell; anyway we still hold that refusing to proceed at any cost in a hysterical paralysis is not the most useful reaction to the discovery that a program might, or even *will*, fail.

4.4 Summary

We have shown a static semantics of ε_0, permitting to statically infer the dimension of the bundle each expression in a static program may evaluate to. We have then proceeded to prove our static analysis to be sound with respect to ε_0's dynamic semantics, providing a sufficient condition which guarantees certain failures not to happen at run time.

Such formal work is practical and not overly complicated, thanks to the minimalistic nature of ε_0.

Dimension analysis can be used for rejecting programs not respecting the sufficient condition; anyway we advocate against such practice, in the interest of extensibility.

Chapter 5

Syntactic extension

The core language ε_0 specified in §2 is useful but inconvenient for humans to write directly. In this chapter we are going to specify syntactic abstraction mechanisms allowing users to easily extend the language by adding high-level syntactic forms to be automatically rewritten into ε_0.

Since the extension facility is defined in ε_0 itself and tightly intertwined with the problem of expressing language syntax as a data structure, we also need to deal with a bootstrapping problem in the process.

Contents

Despite some fundamental differences, the syntactic layer of ε is strongly inspired by Lisp and indeed adopts many conventions taken from Scheme and Common Lisp. We are now proceeding to quickly review Lisp dialects, in order to establish a coherent foundation for our critique.

5.1 Preliminaries

Lisp is a family of dynamically-typed higher-order call-by-value imperative programming languages, suitable to be used in a functional style and particularly convenient for symbolic processing.

The original "LISP" language described by John McCarthy back in 1959[1] [51] has been extended and independently re-implemented many times throughout the years giving birth to a wealth of *dialects*, the most important being the large and complex *Common Lisp* [4] and the elegant, minimalistic *Scheme* [89, 41, 79]. All dialects share the same core ideas.

Contrary to persistent misinformation, most modern Lisps are statically scoped ("*lexically* scoped" in Lisp jargon); Scheme and Common Lisp in particular have

[1] McCarthy specified in a 1995 footnote that [51], published in April 1960, "was written in early 1959": see footnote 4 at page 16 in `http://www-formal.stanford.edu/jmc/recursive.pdf`.

been using static scoping since their original inception in 1975 and 1984.

Lisp introduced several striking innovations most of which eventually found their way into the mainstream, including the interactive Read-Eval-Print Loop, conditional expressions, higher order and garbage collection. Recursion has been supported since the very beginning, and in the 1960s the possibility of expressing a program as a collection of recursive procedures might have felt like the most radical feature. But what still sets apart Lisps from the other languages after fifty years is their *homoiconicity*: programs are encoded using the same data structure they manipulate, which is in fact the only existing "data type" in the language; such data structure, the *s-expression*, is simple and convenient for meta-programming and for representing symbolic information in general.

Just to be explicit from the beginning and to prevent misunderstandings, we already make it clear that ε's syntax will use s-expressions but will *not* be homoiconic: ε is not a Lisp. Yet we find it best to illustrate our solution in an incremental way, starting from a description and critique of Lisp and then re-tracking the mind path by which we arrived at our design.

In the following we give our definition of s-expressions and then proceed to quickly review the main ideas of Lisp, without exactly following any particular dialect. Our lexical and syntactic conventions will mostly come from Scheme, but our macro system will be closer to Common Lisp's.

Our *meta*-linguistic conventions by contrast will be non-standard, particularly with regard to s-expressions, in order to establish a new common framework encompassing ε as well. Experienced Lisp users interested in a comparison with the traditional jargon are referred to the footnotes for some discussion of the rationale for our changes.

5.2 S-expressions

The s-expression is an inductive data structure: it can be seen a disjoint union containing at least several fixed *atomic types* and an *s-cons* type (pronounced "ess-cons"); an *s-cons* is an ordered pair of s-expressions[2].

The specific collection of atoms depends on the Lisp dialect, but at least some types are always provided: a unique object called *the empty list*, *fixnums*, and *symbols*. Symbols are objects identified by a unique name, which can be compared for equality with one another. All dialects also allow *procedures* (zero or more s-expressions as parameters, one s-expression as result, possibly with side effects) as s-expressions.

All practical Lisp dialects also support other atom types, including booleans and other numeric types; other non-atomic types such as vectors are available as well, despite not being required for our presentation.

[2]S-conses are called "conses" in Common Lisp and "pairs" in Scheme.

It is worth to stress the *disjoint-union* nature of s-expressions; however in this slightly non-standard presentation we prefer to explicitly specify an encoding for an s-expression as a pair made of a natural type identifier and an element of the corresponding type.

The following "open-ended" definition is slightly involved due to the nature of s-expressions as a disjoint union whose cases, despite not being all specified, are potentially recursive[3]:

Definition 5.1 (s-expression) *Let $\mathbb{A}_0, ..., \mathbb{A}_{n-1}$ be an ordered collection of sets called addend types including at least the set of fixnums, the set of symbols, the empty list singleton, the set of conses, and some set of s-expressions-to-s-expression procedures.*

We define the set of s-expressions $\mathbb{S}_{\mathbb{A}_0,...,\mathbb{A}_{n-1}}$ (or simply $\mathbb{S}$ without subscripts when the addends are clear from the context) as $(\{0\}\times\mathbb{A}_0)\cup(\{1\}\times\mathbb{A}_1)\cup...\cup(\{n-1\}\times\mathbb{A}_{n-1})$.

For each addend type $\mathbb{A}_i$ we also define:

- *the injected type s-$\mathbb{A}_i$ (pronounced like $\mathbb{A}_i$ preceded by the syllable "ess") as $\{i\} \times \mathbb{A}_i$;*
- *the $\mathbb{A}_i$-injection function $in_{\mathbb{A}_i} : \mathbb{A}_i \to \mathbb{S}$ as $\{x \mapsto (i, x) \mid x \in \mathbb{A}_i\}$;*
- *the $\mathbb{A}_i$-ejection partial function $ej_{\mathbb{A}_i} : \mathbb{S} \rightharpoonup \mathbb{A}_i$ as $\{(i, x) \mapsto x \mid x \in \mathbb{A}_i\}$.*

And the untyped ejection function $ej : \mathbb{S} \to \bigcup_i \mathbb{A}_i$ as $\{(i, x) \mapsto x \mid (i, x) \in \mathbb{S}\}$. □

As per Definition 5.1 we call *s-fixnums*, *s-symbols* and *the empty s-list singleton* the injections of fixnums, symbols, and the empty list singleton into s-expressions. We call *s-conses* the injection of conses — which, it is worth to stress once more, means the set of *s-expression* pairs, rather than *any* pairs. The specific nature of *S-procedures* depends on the Lisp dialect, but in general we can think of them as the injection of procedures with effects accepting zero or more s-expressions and returning one s-expression[4].

Up to this point we have defined s-expressions as a mathematical structure; but since s-expressions are used for input and output, we also need to specify their *written notation* as a reasonably formal syntax. However, to avoid making our notation too heavy, we will not explicitly distinguish between s-expression literals and their corresponding non-injected literals.

[3]The s-cons is not necessarily the only recursive case. We have already hinted at the "vectors" (*s-vectors* for us) supported by all practical Lisps, whose elements are other arbitrary s-expressions; but the idea of course is to enable the user to provide more recursive addends herself.

[4]In our presentation we use the "s-" prefix for explicitly highlighting the difference between a type and its injection into s-expressions, but such distinction is not needed in Lisp where *every* object is an s-expression: s-fixnums, s-symbols and the empty s-list in Lisp are just "fixnums", "symbols" and "the empty list". Here we speak of an s-cons as an (injected) pair *of two s-expressions* rather than two arbitrary objects; since here we do not have much use for non-injected *conses* we can also avoid the issue of conses of non-s-expressions.

Definition 5.2 (s-expression syntax) *Comments start with a semicolon and extend up to the end of the line; all whitespace is otherwise ignored.*

- *We write s-fixnums as strings of one or more digits in radix 10, preceded by an optional sign;*
- *we write the empty s-list as "*`()`*", possibly with whitespace or comments between the open and closed parentheses;*
- *we accept as an s-symbol any sequence of characters not containing spaces, dots, semicolons, quotes, backquotes, commas or parentheses which is not well-formed as an s-fixnum or an s-expression prefix as per Syntactic Convention 5.6*[5]*;*
- *if s_1 and s_2 are s-expressions, then we write their s-cons as "*`(`s_1 `.` s_2`)`*".* □

It should be noticed that s-procedures have no syntax in Definition 5.2: this means that they cannot be directly expressed as literal constants.

Some sample s-fixnums are `1234`, `0`, `+12`, and `-42`; `()` is the empty s-list; all of the following are s-symbols: `a`, `b`, `+`, `this-is-an-s-symbol`, `incr!`, `even?`, `pi/4`, `1-`. Some s-conses are `(1 . 2)`, `(a . ())`, `((a . 3) . 1)`, `((a . b) . (57 . d))`.

Since s-conses allow s-expressions to be nested at any depth, it is convenient to unambiguously name specific substructures:

Definition 5.3 (s-cons selectors) *Let s_1 and s_2 be s-expressions; we say that the s-car of* `(`s_1 `.` s_2`)` *is s_1 and the s-cdr of* `(`s_1 `.` s_2`)` *is s_2.*

By definition, let s-car and s-cdr be s-cons selectors; now, if s-cPr is an s-cons selector for some "path" $P \in \{a, d\}^+$, we define the s-cons selector s-caPr of s to be the s-car of the s-cPr of s, and the s-cons selector s-cdPr of s to be the s-cdr of the s-cPr of s[6]*. When pronounced, each "a" and "d" in s-cons selector names belongs to a different syllable.* □

So for example, the s-caddr (pronounced "ess-ca-dh-dr") of an s-expression is the left element of the right element of its right element, and the s-caddr of `(a . (b . (c . (d . e))))` is `c`.

Apart for their "s-" prefix, introduced by us to distinguish s-expressions from addends, s-cons selector names are well-established in Lisp. They trace back their alien-sounding names to details of the IBM 704, the machine on which McCarthy's LISP was originally implemented [51]; we retain the names for their easy composability, and as a homage to Lisp culture.

Since s-conses may be nested arbitrarily, they can encode linear sequences of any length. Such sequences are conventionally nested on the right:

[5] This description is simplified and idealized compared to what realistic Lisps allow: practical dialects provide escaping mechanisms to even embed *whitespace* within a symbol name.

[6] Again, we prepended the "s-" prefix to the traditional cons selector names.

Definition 5.4 (s-list) *We call an s-expression an s-list*[7] *(pronounced "ess-list") if it is either the empty s-list or an s-cons whose s-cdr is an s-list.*

If an s-list s is empty then we say it has no elements; otherwise we call its elements the s-car of s followed by the elements of the s-cdr of s. □

The following three s-expressions are s-lists: `()`, `(a . ())`, `(a . ((1 . 2) . ()))`; the following three s-expressions are *not* s-lists: `foo`, `(a . b)`, `(a . (b . c))`.

It may be worth stressing that an s-list is allowed to have other s-conses as some or all of its elements, which are not restricted by homogeneity or indeed any constraint on their shape.

S-cons syntax becomes clumsy to use when s-expressions are nested too deeply, hence the need for the following syntactic convention:

Syntactic Convention 5.5 (compact s-expression notation) *An s-cons whose s-cdr is either another s-cons or the empty s-list may optionally be written by both:*

- *omitting the dot;*
- *omitting the parentheses around the s-cdr.* □

For example the last two sample s-lists above may also be written as `(a)` and `(a (1 . 2))`; `(a b . c)` is another way of writing the last sample non-s-list above.

Syntactic Convention 5.5 always applies to the "spine" of s-lists, making them more convenient to write than alternative specular structures nested on the left.

It is easy to convince oneself that, even with Syntactic Convention 5.5, s-expression notation remains unambiguous; in particular we do not need any precedence or associativity rule to parse an s-expression in written form, nor grouping brackets — Far from it, parentheses are a fundamental part of the syntax, and can never be added or removed without changing the denoted data structure.

The following shorthand syntax for s-expressions will be useful later:

Syntactic Convention 5.6 (Lisp s-expression prefixes) *For any s-expression s:*

- `(quote` *s*`)` *may optionally be written as* `'`*s;*
- `(quasiquote` *s*`)` *may optionally be written as* `` ` ``*s;*
- `(unquote` *s*`)` *may optionally be written as* `,`*s;*
- `(unquote-splicing` *s*`)` *may optionally be written as* `,@`*s.*

We say that "`'`", "`` ` ``", "`,`" and "`,@`" are s-expression prefixes. □

[7] What we call s-list here is traditionally known as "list" or "proper list" in Lisp. What we would refer to here as a *non-s-list s-cons* is known in Lisp as a "dotted list" (since Syntactic Convention 5.5 does not apply to the structure spine); somewhat confusingly, Lisp "dotted lists" are not "lists".

s-expression ::=

atom	{ *atom* }
| (*s-expression rest*	{ s-cons(*s-expression*, *rest*) }
| *prefix s-expression*	{ s-cons(lookup(*prefix*), s-cons(*s-expression*, ())) }

rest ::=

)	{ () }
| . *s-expression*)	{ *s-expression* }
| *s-expression rest*	{ s-cons(*s-expression*, *rest*) }

Figure 5.1: S-expressions can be parsed with the attributed LL(1) grammar [3, §§4-5] above, also supporting Syntactic Conventions 5.5 and 5.6. The grammar is simple enough to allow for a hand-coded recursive-descent parser, with no need for generators.
Replacing the second alternative for *s-expression* with "| (*rest* { *rest* }" yields an even simpler grammar which recognizes (. *s*) as an alternate degenerate form of any s-expression *s*, as in fact several Lisp implementations do.

5.3 Lisp syntax

Up to this point we have described the syntax of the s-expression language, without providing any corresponding semantics other than the disjoint-union data structure; even Syntactic Convention 5.6 simply describes a more compact way of writing down some inductive data structures, with no meaning deeper than their shape. But of course the entire point of studying s-expressions is encoding programming language syntax into them; the "s-" prefix indeed stands for "symbolic" in [51], and s-expressions make up the syntax of (a superset of) Lisp forms.

5.3.1 Lisp informal syntax

Here we resist the temptation of formally specifying a mapping from s-expressions to terms of a call-by-value λ-calculus with conditionals and literals; such a definition would depend on the Lisp dialect details and would be either idealized or overcomplicated, without adding much to comprehension in any case.

The following high-level description and the examples below will suffice to provide an intuitive idea:

Syntactic Convention 5.7 (Lisp informal syntax) *Let s_1, s_2 and s_3 be s-expressions.*

- *An s-symbol represents a variable with the same name as its ejection;*
- *an s-fixnum or empty s-list is self-evaluating, which is to say represents itself as a literal constant*[8]*;*

[8] In Scheme the "empty list" () is not considered a valid expression nor interpreted as a literal constant, which forces the user to needlessly quote literal () objects. We consider this an unfortu-

- *an s-cons whose s-car is an s-symbol in a specific set and whose s-cdr has the right shape represents the corresponding syntactic form:*
  - `(if` s_1 s_2 s_3`)` *represents a conditional, of which* s_1 *represents the condition,* s_2 *the "then" branch and* s_3 *the "else" branch;*
  - `(lambda` s_1 `.` s_2`)` *represents an anonymous procedure with the parameter names encoded by* s_1*;* s_2 *represents the sequence of forms in the body; if* s_1 *is an s-list of s-symbols, then the parameters have the same names of its element ejections*[9]*;*
  - `(quote` s_1`)` *represents* s_1 *as a literal constant;*
  - `(quasiquote` s_1`)` *represents* s_1 *as a quasiquoted "mostly-literal" structure: the result is a literal structure equal to* s_1 *except for substructures of the form* `(unquote` s`)` *or* `(unquote-splicing` s`)`*, which represent ordinary non-literal expressions:*
    - *for an* `(unquote` s`)` *substructure, the result of evaluating* s *will replace the substructure in the quasiquoted structure;*
    - *for an* `(unquote-splicing` s`)` *substructure the result of evaluating* s*, which must yield an s-list, will be spliced element by element within the containing s-list in the quasiquoted structure;*
  - `(define` s_1 s_2`)` *when* s_1 *is an s-symbol represents a global definition;* s_2 *represents the expression to be evaluated and whose result will be named;*
  - *...*
  - *an s-cons whose s-car is an s-symbol whose ejection is a macro name represents a user-defined syntactic form;*
- *an s-list whose s-car is not a syntactic form name represents a procedure application: the s-car of the cons represents its operator, and the s-cdr contains an s-list with its zero or more operands.* □

So, for example:

- `57` represents the literal constant `57` (an s-fixnum);
- `a` represents the variable named `a`;
- `'a` represents the literal constant `a` (an s-symbol);
- `(a)` represents the application of a procedure named "`a`", with zero arguments;
- `((a 43))` represents the application of a procedure named "`a`" with one argument, the literal constant `43`; the result, presumably another procedure, is in its turn applied with zero arguments;

nate design mistake (within an otherwise quite beautiful construction), from which to intentionally deviate.

[9] It is not worth the trouble to introduce variadic procedures here, but this wording permits us at least not to arbitrarily exclude them.

- `(if (a b) c d)` represents a two-way conditional expression; if the result of the application of the procedure named "`a`" to the value of the variable named "`b`" is true, then the result is the value of the variable "`c`"; otherwise the result is the value of the variable "`d`";
- `(+ 1 2)` represents the application of a procedure named "`+`" to two arguments, the literal s-fixnums `1` and `2`; no special syntax is needed or available for arithmetic operators, which are considered ordinary procedures: as a consequence of Syntactic Convention 5.7 procedure application syntax is rigidly prefix;
- `'(+ 1 2)` represents the literal constant `(+ 1 2)`, which is an s-cons and in particular an s-list, and also happens to be a valid Lisp expression itself;
- `'(if)` represents the literal constant `(if)`, an ordinary data structure which would not be valid as a Lisp expression;
- `` `(if) `` also represents the literal constant `(if)`;
- `` `(a ,b c) `` represents an s-list of three elements: the s-symbol `a`, the value of the variable `b`, and the s-symbol `c`;
- `` `(a ,@b c) `` represents an s-list of two or more elements: the literal s-symbol `a`, all the elements of the s-list which is the value of the variable `b` (assumed to be an s-list), and finally the literal s-symbol `c`;
- What follows is a reasonable definition of a recursive procedure:

Lisp

```
(define factorial
  (lambda (n)
    (if (= n 0)
      1
      (* n (factorial (- n 1))))))
```

The anonymous procedure is evaluated and then globally named "`factorial`": the procedure has one parameter called "`n`", and its body is a simple conditional: if the result of calling the procedure "`=`" with the parameters `n` and zero is true, then the result is one; otherwise the result is the result of calling "`*`" with two parameters: `n`, and the result of calling `factorial` with the result of calling "`-`" with `n` and one.

Of course a small set of predefined procedures must be provided if we want to perform arbitrary computation on s-expression data: in particular we will need to check whether a given s-expression belongs to an addend type (for example, the `symbol?` procedure returns a true s-expression iff its parameter is an s-symbol), plus constructors and selectors (for example, `cons` returns a new s-cons containing its two parameters; `car` returns the s-car of its parameter, which must be an s-cons); we also need a procedure `eq?` to check whether two given s-symbols are equal.

Given such predefined procedures, it becomes conceptually easy to work on symbolic information, including language transformers and interpreters. [51] contained the first Lisp interpreter written in itself *as an ordinary procedure*, in the space of a couple pages of code.

All realistic Lisps also include some macro facility, usually Turing-complete: macros allow the user to define an s-expression-to-s-expression mapping for rewriting a syntactic form into a combination of already available forms; a macro may be thought of as a Lisp procedure to be automatically applied to all instances of a user-defined form, in some phase prior to execution.

As a simple but not unrealistic example, since global procedure definitions and tests for zero are presumably very common, a user might prefer to be able to write the factorial definition above in a more compact way, as:

Lisp

```
(define-procedure (factorial n)
  (if-zero n
    1
    (* n (factorial (- n 1)))))
```

User-defined forms still follow Lisp syntactic conventions[10]: each use of the new forms `define-procedure` and `if-zero` is encoded as an s-cons whose s-car is the s-symbol uniquely identifying them.

Macros are a form of syntactic abstraction (§1.3.1) allowing to factorize recurring code patterns; it should be obvious that procedural abstraction alone as provided by `lambda` and `define` does *not* suffice to express `define-procedure` and `if-zero`, since their s-expression subcomponents are not necessarily valid to be interpreted as expressions, and in any case they do not follow the call-by-value evaluation strategy of procedures.

As builders of syntax from other pieces of syntax, Lisp macros are a prime example of symbolic computation, and a particularly good use case for quasiquoting.

For example, assuming the three parameters of the macro `if-zero` above to be bound to the formals `discriminand`, `then-branch` and `else-branch`, the macro body might be as simple as `` `(if (= ,discriminand 0) ,then-branch ,else-branch) ``.

5.3.2 Critique

The peculiar syntax of Lisp has always been a polarizing issue for users, either loved or despised with a violent fervor. Without trying to pass our personal opinions on the matter as science, we simply emphasize how powerful macro systems of the kind hinted at above are made possible by s-expressions and homoiconicity.

[10] Of course because of their different role Common Lisp "reader macros" [4, §2.2], a form of extension for the s-expression parser, do not fit our classification; Common Lisp "macros" do.

Syntax aside, some circles also perceive as a problem the apparent lack of efficiency and the strongly dynamic nature of the language, including the glaring absence of static checks.

As controversial topics do, Lisp has generated valid criticism and also plenty of noise with popular slogans, myths and half-truths.

- Lisp has always been used for symbolic processing, its very name standing for "List processing"; many users consider it inherently inefficient out of the field of symbolic computation, because of its very high abstraction level.

  Of course Lisp is far from limited to "lists" (s-lists for us); in fact s-lists are but an s-expression subset, useful in practice but not any more "primitive" than others. More importantly, all practical dialects have also included addend types such as random-access vectors and strings for decades; we avoided them in our presentation of s-expressions simply because such addends are not needed for encoding syntax, and in fact this lack of a homoiconic role might actually contribute to make them less visible — yet, they exist.

  Lisp can be compiled with reasonable efficiency, but some overhead due to its strongly dynamic nature is indeed hard to overcome.

- In particular Lisp is dynamically-typed at its core: there is only one data type, the s-expression. Apart from some runtime tagging and checking cost, the main perceived problem is the difficulty of proving any useful static properties on realistic programs. It is not clear whether the language can be made safer without seriously compromising its expressivity.

  We consider this criticism to be valid.

- Popular claims according to which "Lisp programs are abstract syntax trees" or "Lisp has no syntax" (intended as a positive, negative or neutral remark according to the speaker) can be taken as poetic exaggerations at best.

  Equating valid expressions to ASTs is an oversimplification: in fact most s-expressions do *not* map into valid expressions, and the difference between s-expressions and abstract syntax is relevant in practice. The slogan would be slightly more believable if syntax were encoded as an ML-style sum-of-products type, with its rigid constraints on arity and typing — but that would come with a high cost in extensibility.

  Lisp syntax looks uniform when compared to traditional solutions, but it is not nearly as regular as it could be; for example the two atomic s-expressions `1` and `a` are interpreted in radically different ways, the first as a literal s-fixnum and the second as a variable. A literal s-symbol needs to be quoted as in `'a`, while a literal s-fixnum may be indifferently quoted (once) or not: the s-expressions `1` and `'1` are mapped into the exact same expression. Procedure application syntax is also problematic: an s-expression such as `(a b c)` is regarded as a

procedure call only "as a fallback case", when the s-symbol `a` does not happen to be the name of some syntactic form.

Could Lisp syntax be made more regular? Of course yes: as an alternative we could require form names as explicit s-symbols in the first position of s-lists also for variables and calls, and require quoting for all literals. Then instead of `(* n (f (- n 1)))` we would have something like "`(call (variable *) (variable n) (call (variable f) (call (variable -) (variable n) '1)))`", more uniform but hardly more convenient. Notation would remain clumsy even after introducing new s-expression prefix syntax for "`variable`" and "`call`" in the style of Syntactic Convention 5.6: for example, the very cluttered s-expression "`@($* $n @($f @($- $n '1)))`" is a representation of the expression above, *assuming an s-expression syntax amendment* disallowing "`$`" characters in symbol names — without the syntax change we would need more whitespace, as in "`@($ * $ n @($ f @($ - $ n '1)))`".

Lisp syntax is a compromise and a consequence of conscious design decisions rather than historical accidents, and these issues have been known for decades: [87, "{`FUNCALL` is a pain}", pp. 26-27] already deals with the problem of using "lists" both for procedure calls and for other forms.

We have to recognize that Lisp notation in practice is useful and justified as it stands, despite its relative asymmetry.

5.4 Syntactic extensions: the ε_1 personality

In the following we are going to build upon the experience of Lisp and address all three points in §5.3.2, so that:

- ε be efficiently implementable, and not especially tied to symbolic processing;
- personalities remain open to any typing policy: strong, weak, static, dynamic, hybrid, or none at all;
- ε syntax be at least as convenient as Lisp's while remaining simple to describe and extend.

For extensibility's stake, we use s-expressions to encode language syntax, as Lisp dialects do; but differently from Lisp we choose to decouple syntax and generic data structures, so that s-expressions are available as objects to compute just *as one data type among a wealth of others*: in practice data of each addend type are available either injected into s-expressions (for example s-fixnums, s-symbols), or untagged (fixnums, symbols): thus s-expressions become a way of *selectively* employing dynamic typing in a world where untyped objects are also available, with injection and ejection operators to provide a link between the two representations. S-expressions are always used to represent syntax before macroexpansion, but a user is free to employ them at run time as well if she chooses to, where dynamic typing

feels more convenient. For generality's sake, *we want s-expressions to be extensible* so that the user may provide more addends.

Expressions are just one addend type, *distinct from s-expressions*; ε_0 expressions may be built and analyzed with constructor and selector operators, injected to and ejected from s-expressions. Said even more explicitly, in our solution we have that s-expressions are distinct from *injected expressions*; and macros act like procedures turning s-expressions into untagged expressions. Moving farther from Lisp we will also define *transforms* (§5.4.1.5), as a way of systematically turning (possibly extended) untagged expressions into other untagged expressions.

The personality stack

The language roughly outlined above constitutes a personality we call ε_1:

- The ε_1 *personality* corresponds to ε_0 augmented with forms to define *globals*, *procedures*, *macros* and *transforms*; plus some utility library.
- Thanks to macros and transforms ε_1 is suitable to further extend into higher-level personalities.
- We call ε *the whole system*, including ε_0, ε_1 and other (at this point still hypothetical) higher-level personalities built on top of ε_1.

Higher-level personalities will contain macro and transform definitions in the style of the ones of §5.4.4, later in this chapter.

As a language, ε_1 has an abstraction level between ε_0 and Lisp, closer to the former. Not necessarily aimed at the final user, ε_1 has a low-level feel and is by design unsafe and unforgiving: operators can be applied to the wrong operands with no type checking at all, and pointers are explicit. It lends itself to efficient execution, and is portable if used correctly. ε_1 is compatible with garbage collection but does not require it: the *residual program* resulting when all syntactic abstractions are transformed away might very well use manual memory management only.

The implementation language of ε_1 is ε_0, taking advantage of Scheme for bootstrapping only. The implementation forces us to commit some decisions which we had left open in the description of ε_0 in §2, such as the actual definition of names and handles in terms of data structures. All of this has a bearing on ε_1, and in our solution the implementations of ε_0 and ε_1 are intimately bound: an implementation of ε_0 alone directly parsing the syntax of Definition 2.1, despite being certainly possible, in practice would be little more than an idle exercise without the syntactic extension mechanisms of ε_1.

5.4.1 Definition via bootstrapping

One central idea of ε is to keep the core language as simple as possible, and have more complex linguistic features defined *as code*. As a consequence of this strategy,

a formal specification of ε_0 automatically constitutes a formal[11] specification of ε_1 as well, if we keep into account the source code to bootstrap it from ε_0. Our implementation thus also serves as a specification of ε_1: code, rather than much less flexible mathematics.

The bootstrapping process is nontrivial, and relying as it does on alternative implementations of the same data types, macros, side effects on a global state and *unexec* it provides a particularly poor fit for the graphical notation of T diagrams [52], [39, §3]; here we will resort to plain English to describe the bootstrap phases, and present the source code following by necessity a *bottom-up* style.

The general plan, developed in greater detail throughout the rest of this section, consists of four phases:

(i) extend Scheme by adding untyped data (§5.4.1.1);

(ii) implement ε_0 with s-expression syntax plus definition forms using Scheme macros (§5.4.1.2);

(iii) in this temporary ε_0 implementation, build the core data structures we need upon untyped data, an ε_0 self-interpreter relying on reflective global structures, macros and transforms (§5.4.1.3);

(iv) fill reflective global structures by re-interpreting the core definitions above, so that the interpreter becomes usable (§5.4.1.7);

As it should be clear now, developing ε_1 from ε_0 up to the point where we can define s-expressions, macroexpansion and transforms requires a certain amount of code (about 2000 lines) in which we have to use ε_0 to build some machinery, much of which is useful as part of a generic utility library as well and hence deserves to be considered as "belonging" to ε_1. Part of the "library" in ε_1 exists because of this necessity, while most of the rest relies on syntactic abstraction and is defined *after* the fourth phase, the aim being simply to make ε_1 more convenient to use (§5.4.4).

The fourth phase, after which the global state can be queried, also makes it possible to *unexec* away from Guile into a different runtime (§3.3.2.1).

In the following we are going to show code snippets from the implementation, which is available in a public bzr repository on GNU Savannah: `https://savannah.gnu.org/bzr/?group=epsilon` [2015 note: the repository switched to git in late 2013: see §5.4.5]. We will usually omit or condense comments and may change indentation for reasons of space, but we will *not* simplify the code for this presentation.

This discussion deals with the state of the implementation as of Summer 2012.

5.4.1.1 Phase *(i)*: extend Scheme with untyped data

In order to eventually free ourselves from the dependency on Scheme, we need to define our own data structures which are not based on the predefined version of s-

[11] Of course up to the details we did not describe, such as primitives.

expressions. To simplify debugging and avoid reusing Scheme features *by mistake*, it is also useful to make our data structure incompatible with predefined s-expressions addends; and since we want to unexec in the end, our "untyped" data structure will actually need *boxedness tags* (§3.3.2.1).

We use Guile [21] as our Scheme implementation for bootstrapping. One of the intended applications of Guile is as an embeddable Scheme system to make C applications extensible in the style of Emacs [81], and in view of this use case Guile's C interface was made particularly convenient and flexible; we used it to define in C our new "type" that we call *whatever*, and operations over it. Boxedness tags serve only for the internal Guile garbage-collection machinery, at unexec time, and for debugging memory dumps (§3.3.1); but data structures built with whatevers should be *thought of* as untyped most of the time, as in fact they are conceived for being eventually unexeced into untyped objects, dropping any tagging information.

Whatever operations help to prevent possible mistakes during the bootstrap process by actually performing dynamic checks on tags, in particular to prevent non-whatever objects from being written into whatever buffer slots: whatever data structures must remain *closed over the "points-to" relation*, so that no dependency on Guile s-expressions can remain at unexec time, and instead whatevers only refer other whatevers.

Since Guile is a Scheme implementation its only data type is the s-expression, of which whatevers are seen as just one more addend type: in our extended Guile it is possible to dynamically check whether an s-expression is a *whatever injection*.

The implementation of this phase, mostly in `bootstrap/whatever-guile/whatever-guile.c`, is dirty and not especially interesting in itself. We defined the whatever "type" in C as a *Smob* [21, §Defining New Types (Smobs)]. Whatevers have the printed syntax of Syntactic Convention 3.4, also using ANSI terminal color escape sequences to help the user to recognize boxedness at a glance.

The same C source file also defines operations over whatevers, making them accessible to Scheme: there are trivial *conversion operators* (for example from Scheme (injected) fixnum or (injected) threads to whatever and vice-versa), plus what ε_0 sees as primitives:

- *arithmetic* and *bitwise-logic* operators;
- *memory* allocation, disposing, lookup and update;
- very simple *input/output*;
- *unexecing* primitives, for checking the boxedness tags and buffer sizes;
- the single primitive `state:update-globals-and-procedures!`, needed for transforms (§5.4.1.5).

Primitives number around 30.

The result of this phase is `guile+whatever`, an extended Guile which can also be used interactively, supporting our whatever objects while remaining completely compatible with Scheme. We will not show examples of its use, because some counter-intuitive choices were dictated by efficiency concerns; the details of the `guile+whatever` system become irrelevant anyway after phase *(ii)*.

5.4.1.2 Phase *(ii)*: implement ε_0 in extended Scheme

Our implementation uses a variant of ε_0 in which the grammar of Definition 2.1 is augmented by one more production, for an *indirect call* form:

$$e ::= [\texttt{call-indirect}\ e\ e^*]_h$$

We avoid a formal specification of semantics for `call-indirect`: the idea is simply calling a procedure whose name is computed at run time as the result of an expression; parameters are evaluated call-by-value left-to-right as always in ε_0, first the operator and then the operands.

It is easy to convince oneself that adding `call-indirect` is a quite harmless optimization, as its effect can be easily simulated by automatically generating an "*apply* function" dispatching over one of its parameters, as in Reynolds' defunctionalization [72]. In fact we do that *as well*, as a proof of concept in `bootstrap/scheme/core.e` (§5.4.1.3).

Before we can use ε_0 for implementing ε_1, we need of course a syntax for ε_0. In typical bootstrapping fashion, we would like to define it using the language itself, ε_0 (or maybe ε_1, for maintainability's stake) — but no parser is available. Our solution is mapping an s-expression encoding of ε_0 syntax into Scheme, by using Guile macros[12].

Later we will provide another cleaner frontend implementation[13] in ε_1, to break the bootstrap dependency from Guile; that second frontend will be backward-compatible with this bootstrap implementation of ε_0.

As a consequence of this decision, it is natural for our implementation to use *symbols* for names, encompassing *all* the sets of variable, procedure and primitive names $\mathbb{X}$, $\mathbb{F}$, $\mathbb{\Pi}$.

The following definition, very simple despite its length, follows the spirit of Syntactic Convention 5.7 but is more rigorous due to its importance: reading ε_0 syntax as encoded into s-expressions is key to understand most of the details in the bootstrap process.

Since macros are not supported in this phase but the process is akin to macroexpansion, we name this rewriting of an s-expression into an expression *non-macro*

[12] We used Guile's non-standard Common Lisp-style macros instead of the standard R5RS hygienic macros [41] which Guile also provides. This choice has no particularly deep reason except esthetic consistency with ε_1's macro system; an implementation based on hygienic macros would have worked just as well.

[13] This is not implemented yet: see §5.4.5.

expansion. The choice of generated fresh handles is immaterial in practice, so we speak of non-macro expansion as of a function.

Definition 5.8 (non-macro expansion) *Let s, s_1, s_2, s_3, s_4 be s-expressions. Then we define, up to the choice of a fresh $h' \in \mathbb{H}$, the non-macro expansion function $E_{\mathbb{E}}(\!|\_|\!) : \mathbb{S} \rightharpoonup \mathbb{E}$ as:*

- $E_{\mathbb{E}}(\!|$`(e0:variable` s`)`$|\!) = x_{h'}$ *where* $x = ej_{symbol}(s)$;
- $E_{\mathbb{E}}(\!|$`(e0:value` s`)`$|\!) = c_{h'}$ *where* $c = ej(s)$;
- $E_{\mathbb{E}}(\!|$`(e0:let` s_1 s_2 s_3`)`$|\!) = [$`let` $x_1 ... x_n$ `be` e_{h_1} `in` $e_{h_2}]_{h'}$ *where* $\langle x_1 ... x_n \rangle = E_{\mathbb{X}^*}(\!|s_1|\!)$, $e_{h_1} = E_{\mathbb{E}}(\!|s_2|\!)$ *and* $e_{h_2} = E_{\mathbb{E}}(\!|s_3|\!)$;
- $E_{\mathbb{E}}(\!|$`(e0:call` s_1 `.` s_2`)`$|\!) = [$`call` f $e_{h_1} ... e_{h_n}]_{h'}$ *where* $f = ej_{symbol}(s_1)$ *and* $\langle e_{h_1} ... e_{h_n} \rangle = E_{\mathbb{E}^*}(\!|s_2|\!)$;
- $E_{\mathbb{E}}(\!|$`(e0:call-indirect` s_1 `.` s_2`)`$|\!) = [$`call-indirect` e_{h_0} $e_{h_1} ... e_{h_n}]_{h'}$ *where* $e_{h_0} = E_{\mathbb{E}}(\!|s_1|\!)$ *and* $\langle e_{h_1} ... e_{h_n} \rangle = E_{\mathbb{E}^*}(\!|s_2|\!)$;
- $E_{\mathbb{E}}(\!|$`(e0:primitive` s_1 `.` s_2`)`$|\!) = [$`primitive` π $e_{h_1} ... e_{h_n}]_{h'}$ *where* $\pi = ej_{symbol}(s_1)$ *and* $\langle e_{h_1} ... e_{h_n} \rangle = E_{\mathbb{E}^*}(\!|s_2|\!)$;
- $E_{\mathbb{E}}(\!|$`(e0:if-in` s_1 s_2 s_3 s_4`)`$|\!) = [$`if` $e_{h_1} \in \{c_1 ... c_n\}$ `then` e_{h_2} `else` $e_{h_3}]_{h'}$ *where* $e_{h_1} = E_{\mathbb{E}}(\!|s_1|\!)$, $\langle c_1 ... c_n \rangle = E_{\mathbb{C}^*}(\!|s_2|\!)$, $e_{h_2} = E_{\mathbb{E}}(\!|s_3|\!)$ *and* $e_{h_3} = E_{\mathbb{E}}(\!|s_4|\!)$;
- $E_{\mathbb{E}}(\!|$`(e0:fork` s_1 `.` s_2`)`$|\!) = [$`fork` f $e_{h_1} ... e_{h_n}]_{h'}$ *where* $f = ej_{symbol}(s_1)$ *and* $\langle e_{h_1} ... e_{h_n} \rangle = E_{\mathbb{E}^*}(\!|s_2|\!)$;
- $E_{\mathbb{E}}(\!|$`(e0:join` s`)`$|\!) = [$`join` $e_{h_1}]_{h'}$ *where* $e_{h_1} = E_{\mathbb{E}}(\!|s|\!)$;
- *we do not explicitly specify* $E_{\mathbb{E}}(\!|$`(e1:define . s)`$|\!)$;
- $E_{\mathbb{E}}(\!|$`(e0:bundle . s)`$|\!) = [$`bundle` $e_{h_1} ... e_{h_n}]_{h'}$ *where* $\langle e_{h_1} ... e_{h_n} \rangle = E_{\mathbb{E}^*}(\!|s|\!)$;
- $E_{\mathbb{E}}(\!|s|\!) = E_{\mathbb{E}}(\!|$`(e0:variable` s`)`$|\!)$ *where* s *is an s-symbol;*
- $E_{\mathbb{E}}(\!|(s_1 \text{ . } s_2)|\!) = E_{\mathbb{E}}(\!|$`(e0:call` s_1 `.` s_2`)`$|\!)$ *where* s_1 *is an s-symbol not in* {`e0:variable`, `e0:value`, `e0:let`, `e0:call`, `e0:call-indirect`, `e0:primitive`, `e0:if-in`, `e0:fork`, `e0:join`, `e0:bundle`, `e1:define`};

where the non-macro sequence expander $E_{\mathbb{E}^*}(\!|\_|\!) : \mathbb{S} \rightharpoonup \mathbb{E}^*$ is:

- $E_{\mathbb{E}^*}(\!|$`()`$|\!) = \langle\rangle$;
- $E_{\mathbb{E}^*}(\!|(s_1 \text{ . } s_2)|\!) = e_{h_1}.E_{\mathbb{E}^*}(\!|s_2|\!)$ *where* $e_{h_1} = E_{\mathbb{E}}(\!|s_1|\!)$;

the symbol sequence expander $E_{\mathbb{X}^*}(\!|\_|\!) : \mathbb{S} \rightharpoonup \mathbb{X}^*$ is:

- $E_{\mathbb{X}^*}(\!|$`()`$|\!) = \langle\rangle$;
- $E_{\mathbb{X}^*}(\!|(s_1 \text{ . } s_2)|\!) = x.E_{\mathbb{X}^*}(\!|s_2|\!)$ *where* $x = ej_{symbol}(s_1)$;

and the value sequence expander $E_{\mathbb{C}s}(\!|\_|\!) : \mathbb{S} \rightharpoonup \mathbb{C}^*$ *is:*

- $E_{\mathbb{C}s}(\!|()|\!) = \langle\rangle$;
- $E_{\mathbb{C}s}(\!|(s_1 \ . \ s_2)|\!) = c.E_{\mathbb{C}s}(\!|s_2|\!)$ *where* $c = ej(s_1)$. □

The file `bootstrap/scheme/epsilon0-in-scheme.scm` implements non-macro expansion with Scheme macros. After loading it from `guile+whatever`, Scheme and ε_0 can be used together:

- `(+ 1 2)` yields `3` as a Guile s-expression.
- `(e0:primitive fixnum:+ (e0:value 1) (e0:value 2))` yields the injected whatever `3`, written in green as "3" (§3.3.1).

It is worth remarking how Definition 5.8 does not define any self-evaluating atom, since doing so would create ambiguity with Scheme's predefined self-evaluating atoms: using `e0:value` in cases such as the example above is hence necessary, at this stage: for example `(e0:value 2)` generates `2` as an injected whatever literal constant, which is different from Guile's `2`.

By contrast it is not necessary to use `e0:variable` and `e0:call` for implementing variables and procedure calls, as a consequence of the fact that Scheme and ε_0 share the same namespace for identifiers — at least at this stage.

At this point ε_0 would be usable as an implementation language, if it provided some way of defining procedures and updating the global environment. A correct implementation of such facilities relies on reflective data structures and therefore belongs in Phase *(iii)* or even later; but once more we can use Guile to solve the bootstrap problem and provide a temporary implementation of an `e1:define` form.

As for Scheme's `define`[14], we use the same form for defining either a non-procedure or a procedure, according to the shape of its s-cadr — respectively an s-symbol, or an s-list of one or more s-symbols.

For a non-procedure definition, the second parameter is non-macro expanded, evaluated and the result bound to the symbol-ejection of the first parameter; for a procedure definition, the second parameter is non-macro expanded and bound as the body of a procedure whose name is the symbol-ejection of the s-car of the first parameter; the s-cdr of the first parameter contains an s-list of s-symbols whose ejections make up the procedure formals.

[14] An important difference with respect to Scheme is how our definition facility always works on *state environments* (§2.4.1), therefore at the *top* level, and can be invoked anywhere an expression can occur in the code, at any nesting level. By contrast Scheme's definition facility updates the "current" environment, which happens to be the global one only if the form is used at the top level. Implementing ε's definition form over Guile required a relatively advanced and non-portable hack relying on Guile's module system. See the definition of `define-object-from-anywhere` in `bootstrap/scheme/epsilon0-in-scheme.scm` for the gory details.

Again, `e1:define` is important for understanding the bootstrapping code and deserves a more precise description. Without explicitly specifying a non-macro expansion of an `e1:define` form into an ε_0 expression, we describe the behavior we require from such an expression:

Axiom 5.9 (definition forms) *Let s_1 and s_2 be s-expressions. Then, if $e_h = E_{\mathbb{E}}(\!|$`(e1:define` s_1 s_2`)`$|\!)$:*

- *if $x = ej_{symbol}(s_1)$, $e_{h_2} = E_{\mathbb{E}}(\!|s_2|\!)$ and $e_{h_2}\ \Gamma\ \Downarrow_{\mathbb{E}}\ \langle c\rangle\ \Gamma'$ then we have that $e_h\ \Gamma\ \Downarrow_{\mathbb{E}}\ \langle\rangle\ \Gamma'[^{x\ \mapsto\ c}_{\texttt{global-environment}}]$;*
- *if $\langle f, x_1 ... x_n\rangle = E_{\mathbb{X}_{\mathbb{S}}}(\!|s_1|\!)$ and $e_{h_2} = E_{\mathbb{E}}(\!|s_2|\!)$ then we have that $e_h\ \Gamma\ \Downarrow_{\mathbb{E}}\ \langle\rangle\ \Gamma[^{f\ \mapsto\ (\langle x_1 ... x_n\rangle, e_{h_2})}_{\texttt{procedures}}]$.* □

The two cases are trivially exclusive, as s_1 cannot be an s-symbol and an s-list at the same time.

The reason why we did not provide an explicit non-macro expansion for `e1:define` in Definition 5.8 should be obvious at this point; since the actual implementation is in Scheme and its semantics is very clear, we have avoided writing a uselessly complex expansion into ε_0 assuming some global-updating primitive, even if that would have been possible; in particular it would have been very painful to provide an explicit encoding of ε_0 expressions as ε_0 data; the problem will be dealt with in Phase *(iii)* and in §5.4.4.3, where it becomes relevant for the implementation.

The actual Guile definition of `e1:define` shows an interesting feature: after performing the binding, `e1:define` updates global (Scheme) data structures keeping track of all the ε_0 definitions which have been performed, including procedure bodies. The need for this will become apparent in Phase *(iv)*.

5.4.1.3 Phase *(iii)*: build reflective data structures and interpreter in ε_0

The purpose of this long phase is to define the global reflective data structures holding the program state and then the interpreter. We start from the associated library functionality we need, using only ε_0 in its s-expression encoding and `e1:define`. The task is complicated by the restrictions of the language, allowing for procedural but not syntactic abstraction.

Equipped with Definition 5.8 and Axiom 5.9, the reader should be able to easily follow our running commentary on the main sections of `bootstrap/scheme/core.e`. Each section is delimited by a well-visible comment including a full line of semicolons.

The general low-level "feel" of ε_1 becomes apparent right from the first sections: to create some order in the context of a global flat namespace, we adopt the convention of having all procedure and global names begin with a reasonable *namespace prefix* delimited by a colon. Most of the procedures we define must work also after unexecing, hence they must not rely on unexecing tags primitives: all the code in

`bootstrap/scheme/core.e` works on untyped objects ignoring any boxedness tags; and of course the distinction between booleans, characters or small fixnums is purely conventional; a whatever `0` object may represent the number zero, the false boolean or even a null pointer, according to the context: the machine representation after unexecing is exactly the same.

To make ε_0 more convenient to write, we usually define global procedures to wrap primitives, with their same names; either of the definitions of the *test-for-zero* procedure `whatever:zero?` and the *equality-by-identity* procedure `whatever:eq?` in the first section *Utility procedures working on any data* is a good example:

```
1 (e1:define (whatever:zero? a)
2   (e0:primitive whatever:zero? a))
```

Such definitions only serve to simplify calling syntax, for example allowing the user to write `(whatever:zero?` *s*`)` instead of `(e0:primitive whatever:zero?` *s*`)`; in terms of procedural "abstraction power", they abstract very little.

We start defining operations over the simplest types: the empty list object is simply the fixnum `0`; booleans are represented as physical machines usually do, using `0` for false (written `#f`) and any other value for true, including `1` which we also write as `#t`; in other words, we use *generalized booleans*; the procedure `boolean:canonicalize` canonicalizes a generalized boolean into either `0` or `1`. As a convention derived from Scheme, a question mark "`?`" at the end of a procedure name serves to remind the user that the procedure is a *predicate*, which is to say a procedure returning a boolean result.

The section dealing with *Fixnums* contains primitive wrappers for arithmetic and bitwise operations, plus some very simple definitions such as *minimum, maximum* and a *parity test*; the only slightly more sophisticated procedures are the *fixnum exponentiation* (by squaring) procedure `fixnum:**`, and the base-10 logarithm. An annoying repeating pattern with conditionals is already visible at this point: we often have an `e0:if-in` form testing for a boolean condition, using {`#f`} as the conditional case set to discriminate between false and any other value: what we think of as the "else" branch is always *the first one*:

```
1 (e1:define (fixnum:** base exponent)
2   (e0:if-in exponent (0)
3     (e0:value 1)
4     (e0:if-in (fixnum:odd? exponent) (#f)
5       (fixnum:square (fixnum:** base (fixnum:half exponent)))
6       (fixnum:* base (fixnum:** base (fixnum:1- exponent))))))
```

Unfortunately we cannot factor away this ugly pattern before introducing macros.

The *Buffers* section contains more trivial primitive wrappers for memory-related

primitives: a buffer may be *created* by `buffer:make`, *destroyed* by `buffer:destroy`, *read* by `buffer:get` or *updated* by `buffer:set!`. Buffer size is *only* stored as part of boxedness tags and must not be accessed out of unexec, which is the reason why we did *not* provide a procedure wrapper to conveniently extract it, lest it be used by mistake. `buffer:get` and `buffer:set!` have two and three parameters respectively: we chose to use explicit offsets (0-based, in words) rather than pointer arithmetics for accessing memory, in order to avoid making assumptions on memory management systems, which often constrain the use of inner pointers. As in Scheme, an exclamation mark "`!`" at the end of a procedure name serves to conventionally remind the user that the procedure has side effects.

The *Boxedness* section contains some functionality to check whether a word is a candidate pointer: for example fixnums below a fixed small constant, or fixnums not divisible by the word size in bytes cannot represent pointers on any modern byte-addressed machine (§3.3.2.1).

Having defined buffers, it is very easy to define *Conses*: conses are simply two-word buffers with handy constructor, accessor and updater procedures. Differently from s-conses, conses as defined in this section do not necessarily hold two s-expressions: they are completely generic, mutable *pairs of untyped objects*:

```
1 (e1:define (cons:make car cdr)
2   (e0:let (result) (buffer:make-uninitialized (e0:value 2))
3     (e0:let () (buffer:set! result (e0:value 0) car)
4       (e0:let () (buffer:set! result (e0:value 1) cdr)
5         result))))
```

The nested zero-binding `e0:let` blocks simulating a statement sequence is another unfortunate recurring pattern which we are forced to live with until we introduce macros.

The *Lists* section introduces singly-linked lists made of right-nested conses, and utility procedures to work with them. We "define" a list as either the empty list `list:nil`, which is to say `0`, or a cons whose right element is another list. The quotes in the previous sentence are necessary: in keeping with ε_1's nature the fact that the right side of a list cons be another list is a pure convention, never enforced with static or dynamic checks. Faithful to the motto *"garbage in – garbage out"*, we simply let the system fail at run time when a non-pointer, non-cons or a cons whose right side is not a list is used in place of a list — possibly with a reasonable error message in the case of `guile+whatever`, but likely with a crude *Segmentation Fault* after unexec (§3.3.2.1).

Of course we impose no constraint over the element shape, and lists are not necessarily homogeneous. Our utility procedures over lists include the usual operations for appending, flattening, computing length, selecting by index. Thanks to `e0:call-indirect` we could have supported higher-order procedures (without non-

locals), but we refrained from doing so in this low-level core.

A useful way of employing lists is to make *association lists* or *alists* (pronounced "ey-lists"), the only slightly delicate issue in our case being the way of comparing keys: the first, simplest and most efficient way is "by identity": the section *Alists with unboxed keys* defines procedure using single-word comparison for keys; this is always appropriate for unboxed or unique keys and when the key identity matters, but is not reliable in general with boxed keys where the same content may be replicated in more than one buffer.

A *vector* is a pointer to a buffer with the first element reserved to store the payload element number, which is useful in many contexts where the size of a random-access sequence is not fixed, and we cannot rely on boxedness tags. The section *Vectors* provides procedures to lookup and update vectors, obtain their length, and some other utility operations including append, blit, and conversion to and from list. The procedure `vector:equal-unboxed-elements?` compares two vectors comparing their respective elements by identity, which is the most common case. No bound-checking is performed, and elements are allowed to be heterogeneous. Vectors as defined in this section cannot be resized, as re-allocating them would change their pointer "identity".

The next section deals with *Characters and Strings*: at this level characters are just fixnums and strings are just vectors — which entails the somewhat space-inefficient choice of having each character take one entire word in memory. Yet string support becomes computationally simple, and the wide range of each character suffices for supporting all Unicode code points, at a fixed width. Apart from some trivial I/O, string procedures are just trivial "aliases", or actually wrappers, of vector procedures:

```
1 (e1:define (string:equal? s1 s2)
2   (vector:equal-unboxed-elements? s1 s2))
```

Having defined string support, we are ready to deal with the second kind of association lists, in the *SALists* section: an *salist* (pronounced "ess-ey-list") has strings, or other vectors with elements compared by identity, as keys.

We then have a short section about *Boxes*: a *box*, similarly to an ML `ref`, encapsulates the idea of a mutable memory cell, implemented as a pointer to a single-word buffer. Utility procedures include support for incrementing a mutable counter, in case a box contains a fixnum. For example (omitting the other obvious variant `box:bump-and-get!`):

```
1 (e1:define (box:get-and-bump! box)
2   (e0:let (old-value) (box:get box)
3     (e0:let () (box:set! box (fixnum:1+ old-value))
4       old-value)))
```

Of course `fixnum:1+` is a successor procedure, and `fixnum:1-` is the predecessor[15].

With alists and vectors at our disposal we are ready to implement *Hashes* having either unboxed objects or strings as keys: we also use a box to introduce a level of indirection making a hash easy to resize. A hash table is therefore a box referring a vector; the first payload element of the vector (after the vector "header" word holding the element number) is reserved to keep the hash element number, so that the fill factor is easy to compute at any time; the other vector elements are the hash buckets, implemented as alists or salists; of course the associated data may have any shape. As we need the hash function to be portable with respect to unexec, it respects the constraint in §3.3.2.2. Hashes are our most complex data structure so far: automatic resizing, and particularly comparing fill factors with a threshold *using only fixnums* requires some sophistication, but at around 250 lines our hash table implementation in ε_0 does not end up being overly complicated, despite our choice of avoiding higher order procedures leading to some code redundancy:

```
1  (e1:define (unboxed-hash:set! hash key value)
2    (e0:let () (e0:if-in (hash:overfull? hash) (#f)
3                 (e0:bundle)
4                 (unboxed-hash:enlarge! hash))
5      (unboxed-hash:set-without-resizing! hash key value)))
6  (e1:define (string-hash:set! hash key value)
7    (e0:let () (e0:if-in (hash:overfull? hash) (#f)
8                 (e0:bundle)
9                 (string-hash:enlarge! hash))
10     (string-hash:set-without-resizing! hash key value)))
```

The two-way conditional performing a side effect in one branch and returning a "dummy" bundle in the other is another code pattern which we can't factor away without macros.

Our first and most important application of hash tables is in the implementation of *Symbols*. Since we use them as identifiers, symbols are central in our design; they must be efficient to compare with one another, and as keys in associative structures such as the global (§2.4.2) and procedure (§2.4.4) state environments.

The *symbol table* is a global hash mapping each symbol name, as a string, into a unique boxed *symbol object* with that name; by requiring that all named symbols be *interned* in the table, as all Lisps do, we obtain that symbol pointers can be safely compared by identity, just like unboxed objects; interning the same name more than once yields the same symbol object pointer:

[15]The names "`1+`" and "`1-`" for successor and predecessor come from the Lisp tradition, which we suppose inherited the convention from some Reverse-Polish stack language; to decrement the top stack element in Forth, for example, we may push `1` and then subtract, replacing the two topmost elements with the subtraction result. The Forth code is two words long: "`1 -`". In fact Forth also provides a functionally equivalent (and usually more efficient) predefined single word "`1-`", factoring away the common pattern.

```
(e1:define (symbol:intern name-as-string)
  (e0:if-in (symbol:interned-symbol-name? name-as-string) (#f)
    (symbol:intern-without-checking! (vector:copy name-as-string))
    (string-hash:get symbol:table name-as-string)))

(e1:define (symbol:intern-without-checking! name-as-string)
  (e0:let (new-symbol) (symbol:make-uninterned)
    (e0:let () (buffer:set! new-symbol (e0:value 0) name-as-string)
      (e0:let () (string-hash:set! symbol:table name-as-string new-symbol)
        new-symbol))))
```

Following the example of several Lisps [4, 47] we also support *uninterned symbols*, which is to say symbol objects with no name and hence not occurring in the symbol table; an uninterned symbol pointer can be compared by identity with any other symbol pointer.

The issue of what to store in the symbol object itself appears of little consequence: it is useful to point the symbol name string within the symbol object, to have an efficient mean of retrieving it when needed, typically for printing — but apart from this, the symbol object seems much more useful for its *identity* than for its content. And indeed, were it not for the problem of §3.3.2.2, we could safely use *symbol pointers* like "unboxed" hashed keys for state environments (§2.4.4) such as the global environment or the procedure table; but a better solution has been known for decades, described for example in [82] and in embryonic form already in [51]: the idea is to entirely do away with such global tables whenever possible, and *store the global data associated to each symbol within the symbol object* itself; then the symbol object may be seen as a record, whose fields include the symbol name, the value in the global environment, the formal names of the symbol interpreted as a procedure, the procedure body, and so on. Where a pointer to a symbol is available, accessing it in a state environment only costs one load or store instruction with constant offset. In the interest of extensibility, we also keep one alist field in each symbol object, to which the user is free to add bindings[16].

As a natural consequence of this design, in ε_1 we may use the same symbol as a key for different state environments, for example the global environment and the procedure table, and use the same name for a global non-procedure and a procedure: the possibility of using the same name as key for two (or more) distinct state environments is what distinguishes a so-called *Lisp-2*, such as Common Lisp, from a *Lisp-1*, such as Scheme [32].

The *Symbols* section in the source code would be straightforward except for ε_0's painful lack of records, which at this point we still have to simulate with buffers.

```
(e1:define (symbol:make-uninterned)
  (e0:let (result) (buffer:make (e0:value 9))
```

[16] An alist, later called "property list", was actually the *only* datum globally associated to symbols in [51] (p. 25), also containing a binding for the symbol name. Having fields at fixed offsets in the record object, out of the alist, may be regarded as an optimization.

```
3       (e0:let () (buffer:set! result (e0:value 0) (e0:value 0)) ; name
4         (e0:let () (buffer:set! result (e0:value 1) (e0:value 0)) ; unbound in global environment
5           (e0:let () (buffer:set! result (e0:value 2) (e0:value 127)) ; conventional unbound marker
6             (e0:let () (buffer:set! result (e0:value 3) (e0:value 0)) ; empty formal list
7               (e0:let () (buffer:set! result (e0:value 4) (e0:value 0)) ; no procedure body
8                 (e0:let () (buffer:set! result (e0:value 5) (e0:value 0)) ; no macro definition
9                   (e0:let () (buffer:set! result (e0:value 6) (e0:value 0)) ; no macro procedure
10                    (e0:let () (buffer:set! result (e0:value 7) (e0:value 0)) ; no primitive descriptor
11                      (e0:let () (buffer:set! result (e0:value 8) alist:nil) ; no extensions
12                        result))))))))))
13
14  (e1:define (state:global-set! name value)
15    (e0:let () (buffer:set! name (e0:value 1) (e0:value 1)) ;; the name is bound as a global
16      (buffer:set! name (e0:value 2) value))) ;; value
17  (e1:define (state:procedure-set! name formals body)
18    (e0:let () (buffer:set! name (e0:value 3) formals)
19      (buffer:set! name (e0:value 4) body)))
```

Only from this point on we can find instances of symbol literals in the source, such as `(e0:value foo)`: in the Scheme implementation of ε_0 from Phase *(ii)*, `e0:value` is a Guile macro which generates a whatever object at Scheme macroexpansion time, calling the functionality above for named symbols. The same holds for string literals in the *Strings* section above, but the case of symbols is much more remarkable due to their greater complexity and due to a global structure being involved.

Finally we provide a functionality to automatically generate fresh symbols, particularly useful for machine-generated code. Fresh symbols are interned and have a name starting with a prefix the user is not supposed to use in her own identifiers, currently "`_`". We adopted this solution based on a convention rather than the alternative of using uninterned symbols because of the need to extract *all* symbols from the symbol table (see for example `state:global-names` in §3.1); moreover interned symbols with a conventional prefix are easier to print and read, when needed for debugging. Assuming that "`_`"-prefixed symbols do not occur in user identifiers, generated symbols may be safely garbage-collected[17].

The *Expressions* section defines ε_0 expressions, as per Definition 2.1 plus `call-indirect` (§5.4.1.2), as a data structure. An expression may be conceptually seen as a sum-of-products in the style of ML, and in practice it is implemented as a boxed object: the pointed buffer contains an expression case tag in its first position; the expression handle, present in all cases, resides in the second element; the other element contents, and the buffer length, depend on the specific expression case. Where expressions require homogeneous sequences of undetermined length (for example `bundle` items and `if` conditional cases), by convention we always use lists.

There are operators to build, inspect, update, and *explode* (obtain all compo-

[17] Interned symbols are not yet garbage collected at the time of writing. A solution is employing a second symbol table for for "`_`"-prefixed symbols, implemented as a *weak hash table* [98]. Globals and procedures explicitly named by the user should cannot be safely destroyed in general, as they could be referenced in the future, possibly by dynamically-created expressions.

nents as a bundle) expressions. Such code could conceptually be written by hand, but due to its regularity and length we chose to generate it with a Scheme program; the machine-written ε_0 code is the first non-comment line in the section, easy to spot as the lone, huge 14000-character line with only the minimum required whitespace.

Our expression as managed by the program-generated code has the exact same memory representation as an equivalent data structure defined (much later: see §5.4.4.3) by our general sum-of-product definition facility.

The machine-generated expression constructors require to always specify *all* components, including handles. As the practice is tedious and we can easily generate fresh handles (as fixnums), we also provide a set of hand-written wrappers, named after the symbols identifying the corresponding expression case in expansion followed by an "`*`" character.

```
(e1:define e0:handle-generator-box
  (box:make-initialized (e0:value 0)))
(e1:define (e0:fresh-handle)
  (box:bump-and-get! e0:handle-generator-box))

(e1:define (e0:variable* name)
  (e0:expression-variable-make (e0:fresh-handle) name))
(e1:define (e0:call* name actuals)
  (e0:expression-call-make (e0:fresh-handle) name actuals))
```

For example the ε_0 expression which `(e0:call p 57)` expands to may be built by `(e0:call* (e0:value p) (list:singleton (e0:value* 57)))`, which is not unreadable after considering how the literal constant *expression* which `57` expands to, being an expression, has a different representation from the *fixnum* `57`.

It may be worth to explicitly stress how the ε_0 implementation of Phase *(ii)* does *not* rely at all on the ε_0 expression data structure as defined in this section.

The next section *State: global dynamic state, with reflection* conceptually implements state environments, but for the most part is actually a thin wrapper over buffer accessors used on symbol objects. For example, the following definitions concern the "procedure table", despite it not existing anywhere as a single data structure:

```
(e1:define (state:procedure? name)
  (state:procedure-get-body name)) ;; #f iff unbound, which is to say return the body
(e1:define (state:procedure-get-formals name)
  (buffer:get name (e0:value 3)))
(e1:define (state:procedure-get-body name)
  (buffer:get name (e0:value 4)))
(e1:define (state:procedure-get-in-dimension name)
  (list:length (state:procedure-get-formals name)))
(e1:define (state:procedure-get name)
  (e0:bundle (state:procedure-get-formals name)
             (state:procedure-get-body name)))
```

```
12
13  (e1:define (state:procedure-set! name formals body)
14    (e0:let () (buffer:set! name (e0:value 3) formals)
15      (buffer:set! name (e0:value 4) body)))
16  (e1:define (state:procedure-unset! name)
17    (e0:let () (buffer:set! name (e0:value 3) list:nil)
18      (buffer:set! name (e0:value 4) (e0:value 0)))) ;; make the body invalid
```

It is obvious at this point that our implementation of `e1:define` from Phase *(ii)* could *not* update ε_1's state environments such as the global environment and the procedure table, which we have just implemented: up to this point anything which has been globally defined in Phase *(iii)* has been only set in *Guile's* state environments. This problem will persist until Phase *(iv)*.

By examining all the buckets in the symbol table it is easy to obtain the list of all interned symbols bound to a procedure, or a primitive. From this information we can automatically build an *apply function* in the style of [72], plus another procedure in the same spirit for primitives, without resorting to indirect calls. The automatically-generated procedures `state:apply` and `state:apply-primitive` (collectively *appliers*) each take two arguments, a symbol naming the object to call, and a parameter list. The generated applier body consist in a deeply-nested conditional, comparing the first parameter with each applicable name: when the matching name is found the applier calls the corresponding procedure or primitive, returning its results.

Generating appliers is our first example of dynamically-generated code in the implementation. Of course such generation heavily relies on dynamically-built ε_0 expressions.

Armed with global tables we are finally ready to implement a working interpreter in the next section, *epsilon0 self-interpreter*.

The interpreter code is not overly complex and the main procedure `e0:eval` is but a long dispatcher, selecting the appropriate expression case and tail-calling a helper procedure. Its second parameter is the local environment, encoded as an alist:

```
1   (e1:define (e0:eval e local)
2     (e0:if-in (e0:expression-variable? e) (#t)
3       (e0:let (h name) (e0:expression-variable-explode e)
4         (e0:eval-variable name local))
5       (e0:if-in (e0:expression-value? e) (#t)
6         (e0:let (h content) (e0:expression-value-explode e)
7           (e0:eval-value content))
8         (e0:if-in (e0:expression-bundle? e) (#t)
9           (e0:let (h items) (e0:expression-bundle-explode e)
10            (e0:eval-bundle items local))
11          (e0:if-in (e0:expression-primitive? e) (#t)
12            (e0:let (h name actuals) (e0:expression-primitive-explode e)
13              (e0:eval-primitive name actuals local))
14            (e0:if-in (e0:expression-let? e) (#t)
15              (e0:let (h bound-variables bound-expression body) (e0:expression-let-explode e)
```

```
16              (e0:eval-let bound-variables bound-expression body local))
17            (e0:if-in (e0:expression-call? e) (#t)
18              (e0:let (h name actuals) (e0:expression-call-explode e)
19                (e0:eval-call name actuals local))
20              (e0:if-in (e0:expression-call-indirect? e) (#t)
21                (e0:let (h procedure-expression actuals) (e0:expression-call-indirect-explode e)
22                  (e0:eval-call-indirect procedure-expression actuals local))
23                (e0:if-in (e0:expression-if-in? e) (#t)
24                  (e0:let (h discriminand values then-branch else-branch)
25                          (e0:expression-if-in-explode e)
26                    (e0:eval-if-in discriminand values then-branch else-branch local))
27                  (e0:if-in (e0:expression-fork? e) (#t)
28                    (e0:let (h name actuals) (e0:expression-fork-explode e)
29                      (e0:eval-fork name actuals local))
30                    (e0:if-in (e0:expression-join? e) (#t)
31                      (e0:let (h future) (e0:expression-join-explode e)
32                        (e0:eval-join future local))
33                      (e0:if-in (e0:expression-extension? e) (#t)
34                        (e0:let (h name subexpressions) (e0:expression-extension-explode e)
35                          (e0:eval-extension name subexpressions local))
36                        (e1:error (e0:value "impossible")))))))))))))))
```

In order to keep the code understandable despite the deeply-nested conditionals we chose *not* to assume generalized booleans in `e0:eval`, making sure that all the predicates we used only return `#t` or `#f`.

Several helper procedures in their turn rely on `e0:eval-expressions`, which sequentially evaluates a *list* of expressions which have to return 1-dimension bundles, and returns the result list.

Many interpreter procedures are strongly interdependent and mutually recursive, which is served quite well by procedural abstraction. It is very convenient to define mutually-recursive procedures without concern for the definition order, so that the programmer does not need to keep a call graph in her mind.

```
1  (e1:define (e0:eval-expressions expressions local)
2    (e0:if-in expressions (0)
3      list:nil
4      (list:cons (e0:unbundle (e0:eval (list:head expressions) local))
5                 (e0:eval-expressions (list:tail expressions) local))))
6  (e1:define (e0:unbundle bundle)
7    (e0:if-in (list:null? bundle) (#f)
8      (e0:if-in (list:null? (list:tail bundle)) (#f)
9        (e1:error (e0:value "e0:unbundle: the bundle has at least two elements"))
10       (list:head bundle))
11     (e1:error (e0:value "e0:unbundle: empty bundle"))))
```

Most helper procedures dealing with specific expression cases end up being simple:

```
(e1:define (e0:eval-variable name local)
  (list:singleton (e0:if-in (alist:has? local name) (#f)
                    (state:global-get name)
                    (alist:lookup local name))))

(e1:define (e0:eval-value content)
  (list:singleton content))

(e1:define (e0:eval-if-in discriminand values then-branch else-branch local)
  (e0:let (discriminand-value) (e0:unbundle (e0:eval discriminand local))
    (e0:if-in (list:memq discriminand-value values) (#f)
      (e0:eval else-branch local)
      (e0:eval then-branch local))))

(e1:define (e0:eval-bundle items local)
  (e0:eval-expressions items local))
```

A possibly striking implementation choice consists in encoding ε_0 bundles as lists; this is necessary for examining bundle results, for example by testing their length — the fact that bundles are not denotable (§2.1.5) makes them hard to deal with directly, in exchange for their potential efficiency in a compiled implementation. But ironically in this self-interpreter, where however performance is not a priority, the need of building lists for bundles entails a high rate of heap allocation, which is expensive.

The self-interpreter does not rely on explicit stacks and is quite far from the semantics in §2.5; yet, of course without hope of certifying the implementation *here*, we claim that we believe it respects the semantics and implementation notes of §2.

With the most fundamental addend types at our disposal we are ready to deal with general support for user-defined types, in the *Type table* section.

We start with building support for tracking the extensible set of "types" recognized by the system, such as the empty list, booleans, fixnums, and conses; since in this context we assume dynamic typing, there is no need for type parameters: all the subcomponents of an s-expressions are tagged, at any level.

As types tend to be relatively few in number and this reflective information is not particularly critical to performance, in this case we preferred a global table to the alternative of reserving fields in *all* symbol objects.

Information for each type (empty list, boolean, fixnum, cons, string, ...) is encoded in a descriptor record implemented as a buffer, also containing a unique tag, other information including a printer procedure, and once again an alist which the user may employ to add more fields as type-dependent attributes — it is unfortunately too early to define a general-purpose "extensible record" data structure, without any support for syntactic abstraction.

The most interesting fields in type descriptor records is the *expression-expander* procedure name. An expression-expander specifies how to turn an object of the

given type into an *expression*. Since of course we provide procedures to update the type table, the user has the power to define and change the way addends expand, including, for example the mapping from a symbol into a variable as discussed in Syntactic Convention 5.7 and §5.3.2.

All support for macros, following Lisp syntactic conventions[18], will be defined later in procedures called in its turn by the *cons expander* procedure; our predefined expanders will implement expansion in a way compatible with Definition 5.8.

Definition 5.10 (expression-expander) *Let* $\mathbb{A}_0, ..., \mathbb{A}_{n-1}$ *be the addend types of* $\mathbb{S}$. *Then the expander procedure for* $\mathbb{A}_i$ *or* $\mathbb{A}_i$*-expander is a procedure of one parameter returning one result. The procedure is guaranteed not to fail if the parameter has type* $\mathbb{A}_i$, *in which case the result has type* $\mathbb{E}$. □

Most atomic objects such as the empty list, booleans and fixnums, are expression-expanded by `sexpression:literal-expression-expander` into a literal constant expression, which will finally allow the user to omit explicit "`e0:value`"s for non-symbol literals; `sexpression:variable-expression-expander` expression-expands a symbol into a variable expression; `sexpression:expression-expression-expander` trivially expression-expands an expression into itself. Cutting away some comments:

```
1 (e1:define (sexpression:literal-expression-expander whatever)
2   (e0:value* whatever))
3 (e1:define (sexpression:variable-expression-expander symbol)
4   (e0:variable* symbol))
5 (e1:define (sexpression:expression-expression-expander expression)
6   expression)
```

The cons expression expander is not complicated either, and resembles Syntactic Convention 5.7 (p. 78) in regarding the procedure call as a "default case". It can be already seen from this code how *even* ε_0 *syntactic forms are implemented as macros*:

```
1 (e1:define (sexpression:cons-expression-expander cons)
2   (e0:let (car-sexpression) (cons:car cons)
3     (e0:if-in (sexpression:symbol? car-sexpression) (#f)
4       (e1:error (e0:value "cons:expression-expander: the car is not a symbol"))
5       (e0:let (car-symbol) (sexpression:eject-symbol car-sexpression)
6         (e0:if-in (state:macro? car-symbol) (#f)
7           ;; The car is a symbol which is not a macro name:
8           (e0:call* car-symbol (e1:macroexpand-sexpressions (cons:cdr cons)))
9           (e1:macroexpand-macro-call car-symbol (cons:cdr cons)))))))
```

Lines 3-4 show how the specific cons-expander above is not suitable for higher-order personalities where the operator can be encoded by an s-expression different from an s-symbol. Of course the user is free to replace the cons-expander at a later time.

[18] Common Lisp also supports "symbol macros" [4]: some symbols defined by the user are macroexpanded like zero-parameter macro calls. Support for a similar feature can be added in ε_1 by changing the *symbol*, rather than cons, expander.

The *S-expressions* section deals with the implementation of s-expressions as data structures, and operations over them. Some of our procedures defined up to this point already *call* procedures working over s-expressions such as `sexpression:symbol?`, `sexpression:eject` and `sexpression:inject-cons` in procedure bodies — although of course our procedures calling not-yet-defined procedures have never been called themselves, yet.

The specific memory representation of an s-expression object has always been seen considered important for efficiency in Lisp: all practical Lisps employ some form of bitwise tagging of unboxed objects, boxed pointers and/or buffer words, allowing to store in a compact way elements of the most common addends; such representation techniques are often complex (see [21, §Data Representation] for Guile's solution and [35] as a useful collection of "a body of folklore"), but such complexity is motivated by the need for tagging *all*[19] data in Lisp.

However the situation of ε_1 is quite different from Lisp: s-expressions are mostly used for representing user syntax before macroexpansion, but not necessarily as a data structure after macroexpansion. Even if an efficient implementation is certainly possible (potentially by machine generation as in [75]) for the time being we make do with a quite literal implementation of Definition 5.1: we represent an s-expression as a pointer to a two-element buffer, whose first cell holds the type tag while the second holds the representation of the addend-type content. Some sample definitions:

```
1  (e1:define (sexpression:make tag value)
2    (cons:make tag value))
3  (e1:define (sexpression:get-tag sexpression)
4    (cons:get-car sexpression))
5  (e1:define (sexpression:eject sexpression)
6    (cons:get-cdr sexpression))
7  (e1:define (sexpression:has-tag? x tag)
8    (whatever:eq? (sexpression:get-tag x) tag))
9
10 ;; We have generated unique tags when adding entries to the type table
11 (e1:define (sexpression:null? x)
12   (sexpression:has-tag? x sexpression:empty-list-tag))
13 (e1:define (sexpression:boolean? x)
14   (sexpression:has-tag? x sexpression:boolean-tag))
15 (e1:define (sexpression:cons? x)
16   (sexpression:has-tag? x sexpression:cons-tag))
17
18 (e1:define (sexpression:inject-fixnum x)
19   (sexpression:make sexpression:fixnum-tag x))
20 (e1:define (sexpression:eject-fixnum x)
21   (e0:if-in (sexpression:fixnum? x) (#f)
22     (e1:error (e0:value "sexpression:eject-fixnum: not a fixnum"))
23     (sexpression:eject x)))
```

[19] Advanced optimizing Lisp compilers such as SBCL [73] may actually avoid run-time tagging at some code points, in cases when a type inference analysis succeeds and in favorable contexts. This optimization, however, is not possible for the bulk of the code.

```
24
25 (e1:define sexpression:nil ;; an empty s-list object
26   (sexpression:make sexpression:empty-list-tag empty-list:empty-list))
27 (e1:define (sexpression:car x)
28   (e0:if-in (sexpression:cons? x) (#f)
29     (e1:error (e0:value "sexpression:car: not a cons"))
30     (cons:get-car (sexpression:eject x))))
31
32 (e1:define (sexpression:cadr x) (sexpression:car (sexpression:cdr x)))
33 (e1:define (sexpression:caadr x) (sexpression:car (sexpression:cadr x)))
```

In our representation *all* s-expressions are boxed, and even traditionally "unique" objects such as the empty s-list or s-booleans may exist in more than one instance.

We also define alternate versions of some procedures over fixnums and lists suitable to work on s-fixnums and s-lists, as it will be convenient later in macros to manipulate s-expressions without explicit injections and ejections:

```
1  (e1:define (sexpression:1+ x)
2    (sexpression:inject-fixnum (fixnum:1+ (sexpression:eject-fixnum x))))
3
4  (e1:define (sexpression:reverse x)
5    (sexpression:append-reversed x sexpression:nil))
6  (e1:define (sexpression:append-reversed x y)
7    (e0:if-in (sexpression:null? x) (#f)
8      (sexpression:append-reversed (sexpression:cdr x)
9                                   (sexpression:cons (sexpression:car x) y))
10     y))
```

5.4.1.4 Macros

Still following the bootstrap code in `core.e`, we are now finally ready to add support for *Macros*.

The `e1:macroexpand`[20] procedure, turning an s-expression into a corresponding expression, is but a trivial dispatcher tail-calling the appropriate expression-expander; but some expanders, as by the default the one for cons does, may involve expanding actual macro calls.

```
1  (e1:define (e1:macroexpand s)
2    (e0:let (tag) (sexpression:get-tag s)
3      (e0:let (content) (sexpression:eject s)
4        (e0:call-indirect (sexpression:type-tag->expression-expander-procedure-name tag)
5                          content))))
```

The general idea of macros is simple enough[21]: the user defines each macro "in concrete syntax" as an *s-expression*, often relying on other macros. Before a macro

[20] The name "`macroexpand`" may not be entirely appropriate, but has long been traditional in Lisp circles. Even an s-expression containing no macro calls can be successfully "macroexpanded".

[21] Our mechanism is in practice not unlike the Common Lisp or Emacs Lisp macro systems, despite our explicit distinction between expressions and s-expressions. Common Lisp uses an

call can be expanded the macro body itself must have been macroexpanded in its turn into an *expression*, which makes up the body of the associated *macro procedure*. At macro call time, the macro procedure is called by supplying it with the macro actuals; the macro procedure result, an *s-expression*, is then macroexpanded in its turn, which may involve expanding other macro calls. If the process does not diverge, the final result will be an *expression*. Since predefined macros allow to express all ε_0 forms our macro system is trivially Turing-complete, already because of macro procedures. Of course it is permitted, and useful, for a macro to return another macro call: this allows to build upon user-defined forms, "stacking" syntactic abstractions one onto another.

Macro definition and lookup are easy enough, based as they are is on symbol objects similarly to the global and procedure state environments:

```
1  (e1:define (state:macro-set! macro-name macro-body-sexpression)
2    (e0:let ()
3      ;; If we're re-defining an existing macro, invalidate its previous procedure:
4      (e0:if-in (buffer:get macro-name (e0:value 5)) (0)
5        (e0:bundle)
6        (state:invalidate-macro-procedure-name-cache-of! macro-name))
7      (buffer:set! macro-name (e0:value 5) macro-body-sexpression)))
8  (e1:define (state:macro-get-body macro-name)
9    (buffer:get macro-name (e0:value 5)))
10 (e1:define (state:macro? name)
11   (state:macro-get-body name)) ;; 0 iff unbound, which is to say return the body
```

The careful reader may have noticed a small difference in `state:macro-set!` compared to the analogous code for procedures: no formal parameters names are provided for macros. This absence is a conscious choice of ours, leading to a small simplification: as no nonlocal is ever visible from a macro body, parameter shadowing is impossible and we can safely use the same formal name "`arguments`" for *all* macros. Moreover we can use *one* formal for *all* the parameters of a macro call by viewing them as an s-expression, which is to say the s-cdr of the macro call s-expression — for example `(m a (324) 3)` has parameters `(a (324) 3)`.

Of course we will also add support for friendlier macros with named formals, later as a syntactic extension.

As a concession to efficiency we *cache* macro procedures, by generating them at the time of the first expansion of a macro call and then re-using them. It is important to specify this point, because in rare cases caching may have a observable effect on the result — that is the case of macros performing side effects very early, while building the returned expression.

The corresponding code is surprisingly simple, ignoring the references to transformations for the time being:

auxiliary procedure "`macroexpand-1`" returning two results: the result of expanding *one* call, and a boolean saying whether expansion should continue. In our case we can simply use expression-expanders, which in the terminal case will receive an injected expression.

```
(e1:define (state:macro-get-macro-procedure-name macro-name)
  (e0:let (cached-macro-procedure-name-or-zero) (buffer:get macro-name (e0:value 6))
    (e0:if-in cached-macro-procedure-name-or-zero (0)
      (state:macro-get-macro-procedure-name-ignoring-cache macro-name)
      cached-macro-procedure-name-or-zero)))

(e1:define (state:macro-get-macro-procedure-name-ignoring-cache macro-name)
  (e0:let (body-as-sexpression) (state:macro-get-body macro-name)
    (e0:let (untransformed-name) (symbol:fresh)
      (e0:let (untransformed-formals) (list:singleton (e0:value arguments))
        (e0:let (untransformed-body) (e1:macroexpand body-as-sexpression)
          (e0:let () (state:procedure-set! untransformed-name
                                           untransformed-formals
                                           untransformed-body)
            (e0:let (transformed-name transformed-formals transformed-body)
                    (transform:transform-procedure untransformed-name
                                                   untransformed-formals
                                                   untransformed-body)
              (e0:let () (state:procedure-set! transformed-name
                                               transformed-formals
                                               transformed-body)
                (e0:let () (buffer:set! macro-name (e0:value 6) transformed-name)
                  transformed-name)))))))))
```

A macro call expansion consists in macroexpanding one call into an *s-expression* and then tail-calling to a further macroexpansion of the result, which will hopefully terminate; the usual terminal case is an injected expression.

```
(e1:define (e1:macroexpand-1-macro-call symbol arguments)
  (e0:let (macro-procedure-name) (state:macro-get-macro-procedure-name symbol)
    (e0:call-indirect macro-procedure-name arguments)))

(e1:define (e1:macroexpand-macro-call symbol arguments)
  (e0:let (sexpression-after-one-expansion) (e1:macroexpand-1-macro-call symbol arguments)
    (e1:macroexpand sexpression-after-one-expansion)))
```

Just for completeness we also show the trivial helper, called by the cons expander, which macroexpands an s-list of s-expressions into a list of expressions, left-to-right:

```
(e1:define (e1:macroexpand-sexpressions sexpressions)
  (e0:if-in (sexpression:null? sexpressions) (#f)
    (list:cons (e1:macroexpand (sexpression:car sexpressions))
               (e1:macroexpand-sexpressions (sexpression:cdr sexpressions)))
    list:nil))
```

We close by developing an illustrative and we hope not too artificial example. Let us assume to have somehow added an `e1:trivial-define-macro` form for globally defining a macro, internally using `state:macro-set!`; `e1:trivial-define-macro` has two parameters: the macro name, and the macro body s-expression. We use `e1:trivial-define-macro` to define our sample macro:

```
(e1:define (sexpression:list3 a b c)
  (sexpression:cons a (sexpression:cons b (sexpression:cons c sexpression:nil))))

(e1:trivial-define-macro silly-square
  ;; We would write `(fixnum:* ,(sexpression:car arguments)
  ;;                           ,(sexpression:car arguments))
  ;; if we already had quasiquoting.
  (sexpression:list3 (sexpression:inject-symbol (e0:value fixnum:*))
                     (sexpression:car arguments)
                     (sexpression:car arguments)))
```

The `silly-square` macro takes at least one parameter (ignoring any one after the first) and returns an expression multiplying the parameter by itself; the resulting expression will contain two copies of the macroexpanded parameter, which therefore will be *evaluated twice*.

For example, `(silly-square 4 5 6)` would eventually macroexpand to [`call fixnum:*` $4_{h'_2}$ $4_{h'_3}$]$_{h'_1}$, for some fresh $h'_1, h'_2, h'_3 \in \mathbb{H}$.

When calling `e1:macroexpand` on `(silly-square 4 5 6)` we immediately go through the cons expression-expander; assuming that `silly-square` is *not* a procedure name, we tail-call `e1:macroexpand-macro-call` with two parameters: the symbol `silly-square`, and the s-list `(4 5 6)`; `e1:macroexpand-macro-call` attempts to expand the first call by using `e1:macroexpand-1-macro-call`. Assuming `silly-square` has not been used before, `state:macro-get-macro-procedure-name` builds its macro procedure, which requires several expression-expansion calls not involving macros; `state:macro-get-macro-procedure-name` then returns the macro procedure name to `e1:macroexpand-1-macro-call`, which calls it on `(4 5 6)`; the result is the s-expression `(fixnum:* 4 4)`, which is returned by `e1:macroexpand-1-macro-call`; so control goes back to `e1:macroexpand-macro-call`, which tail-calls `e1:macroexpand` on `(fixnum:* 4 4)`; by trivial expression-expansions, we finally obtain the expression [`call fixnum:*` $4_{h'_2}$ $4_{h'_3}$]$_{h'_1}$.

5.4.1.5 Transforms

Many mathematical presentations deal with "transformations", meant as code-to-code functions. Our *transform* strategy adopts the same approach, with the sole significant extension of also permitting side-effecting procedures.

When building a transform, the user or personality developer has to simply define ordinary procedures working on code, and then to "hook" them to the system. There are two reasonable ways of running such *transform procedures*:

- a procedure can be applied *retroactively* to the current state, adding (and usually replacing) definitions;
- or it can be *installed*, to be applied automatically in the future to each toplevel expression or each procedure created from that point on; since the composition order *is* usually significant, the user can control where each transform procedure fits in a global list of names.

In general we are interested in transforming three different entities: *expressions*, *procedure bindings*, and *global bindings*. Transform procedures will need to be different for each case, since the procedure interface cannot be compatible; but in our experience, "companion" transform procedures tend to rely on some common helper doing most of the actual work: for example, closure-converting a procedure binding involves closure-converting its body, which is an expression (§5.4.4.4).

Global bindings are difficult to work with in practice, since they contain already-evaluated values with no fixed shape, rather than expressions; we have not yet found a use for global binding transform procedures, but we include such support for symmetry reasons.

For generality's sake, we decided to have *binding transform procedures* also return a transformed name: this may be either the same as the original untransformed binding, or a new one. It might be convenient to keep the old definition around for debugging reasons, for example, but change all the uses of the old entity to new one, by systematically renaming references.

A transform procedure may have one of three different interfaces:

- *one parameter – one result*, to transform an expression;
- *three parameters – three results*, to transform a procedure binding: name, formals, body;
- *two parameters – two results*, to transform a global binding: name, value.

The ultimate purpose of our code-rewriting system is to let the user write in an expressive high-level language, to be then automatically reduced to ε_0 by "transforming away" extensions. The transform procedures mapping "syntax into syntax" therefore will need to support not only syntax, but *extended syntax* as data. We will show an elegant solution to this problem in §5.4.4.3, but we do not need to be concerned with it now while discussing the code which *invokes* procedure transforms.

As a less obvious consequence of our design, side-effecting transforms provide for another interesting opportunity: a simple way of performing a *code analysis* is to implement a trivial transform procedure returning its parameters unchanged *while recoding data in some global structure*, possibly a global table with handles (§2.1.3) as keys. Transforms actually returning modified code might also store their parameters somewhere or simply record the relation between transformed an untransformed code[22] in some global structure, available for later debugging or analysis.

Transforms are also a convenient way to run some *optimizations* rewriting expressions into more efficient versions. As a first "low-hanging fruit" we plan to use some *heuristic search* algorithm such as hill-climbing to search the neighborhood of

[22] A practical problem of the current implementation which makes debugging difficult is the lack of a reverse mapping from code to its original untransformed form and ultimately to its the s-expression concrete syntax and source location. Solving this problem requires some care in writing transform, parsing and expression-expansion procedures, so that when an expression is built from others, its origin is somehow recorded in a graph. A linguistic extension to somehow automate this tracing process might be appropriate.

an expression for operationally-equivalent but faster versions.

We are finally ready to present some of the code in the *Transforms* section. At around 150 lines the code is quite short and also very uniform, often present in three just slightly different versions because of the three entities to manage.

The global lists of transform names to be applied in order are simple boxed global variables:

```
(e1:define transform:expression-transforms (box:make-initialized list:nil))
(e1:define transform:procedure-transforms (box:make-initialized list:nil))
(e1:define transform:global-transforms (box:make-initialized list:nil))

(e1:define (transform:prepend-expression-transform! new-transform-name)
  (box:set! transform:expression-transforms
            (list:cons new-transform-name (box:get transform:expression-transforms))))
(e1:define (transform:append-procedure-transform! new-transform-name)
  (e0:let () (box:set! transform:procedure-transforms
                       (list:append2 (box:get transform:procedure-transforms)
                                     (list:singleton new-transform-name)))
    (state:invalidate-macro-procedure-name-cache!))) ;; All macros have to be re-transformed
```

The interaction with macros is interesting as it reminds us that an untransformed procedure may be incompatible with its transformed version (for example, in a CPS transform the argument number may change): it is hence important to invalidate any cached macro procedure, so that new ones are created, *and subjected to the current transforms*.

Applying transform procedures is trivial; this is the code which gets executed when a procedure binding is transformed; the other two cases are essentially identical.

```
(e1:define (transform:transform-procedure name formals body)
  (e0:let (transform-names) (box:get transform:procedure-transforms)
    (transform:apply-procedure-transforms transform-names name formals body)))
(e1:define (transform:apply-procedure-transforms remaining-transforms name formals body)
  (e0:if-in remaining-transforms (0)
    (e0:bundle name formals body)
    (e0:let (transformed-name transformed-formals transformed-body)
            (e0:call-indirect (list:head remaining-transforms) name formals body)
      (transform:apply-procedure-transforms (list:tail remaining-transforms)
                                            transformed-name
                                            transformed-formals
                                            transformed-body))))
```

Retroactive transformation is more interesting. The user will call `transform:transform-retroactively!` to install transform procedures for global and procedure bindings, also specifying the names of some objects *not* to transform.

```
(e1:define (transform:transform-retroactively! globals-not-to-transform
                                               value-transform-names
```

```
3                                                    procedures-not-to-transform
4                                                    procedure-transform-names)
5   (e0:let (global-names) (list:without-list (state:global-names) globals-not-to-transform)
6     (e0:let (procedure-names)
7             (list:without-list (state:procedure-names) procedures-not-to-transform)
8       (e0:let (transformed-name-global-list)
9               (transform:compute-transformed-globals global-names value-transform-names)
10        (e0:let (transformed-name-formal-body-list)
11                (transform:compute-transformed-procedures procedure-names
12                                                          procedure-transform-names)
13          (e0:primitive state:update-globals-and-procedures! transformed-name-global-list
14                                                             transformed-name-formal-body-list))))))
```

The code works by first *computing* all transformed bindings (the trivial helpers `transform:compute-transformed-globals` and `transform:compute-transformed-procedures` simply return a list of transformed bindings) without performing any global update; then, *with a single primitive call*, it activates all new bindings.

Why having such a complex primitive written in C? And why do we have to compute all the bindings before applying any? The answer is that *we need the state environment update to be performed atomically*[23], again because of the incompatibility introduced by some transforms. The alternative of updating global definitions in ε_0 would fail when at some point *the updater procedure itself* or its helpers would be reached by the incompatible *change wave*, and break on a call from an untransformed procedure to a transformed one, or vice-versa. For this reason only, `state:update-globals-and-procedures!` must be a primitive.

The *REPL* section is the last interesting part of `core.e`. Its helper procedure `repl:macroexpand-transform-and-execute` can be given an s-expression to expression-expand, transform and evaluate:

```
1  (e1:define (repl:macroexpand-transform-and-execute sexpression)
2    (e0:let (untransformed-expression) (e1:macroexpand sexpression)
3      (e0:let (transformed-expression) (transform:transform-expression untransformed-expression)
4        (e0:eval-ee transformed-expression))))
```

The REPL itself is very crude, and currently relies on a primitive `io:read-sexpression` *calling Scheme from C* to read a Guile s-expression and then convert it into our representation. This lack of a real frontend written in ε_1 is the last remaining reason why we still depend on Guile after bootstrap (§5.4.5).

```
1  (e1:define (repl:repl)
2    (e0:let () (string:write "Welcome to the epsilon REPL\n")
3      (repl:loop (io:standard-input))))
4  (e1:define (repl:loop port)
```

[23] Nothing to do with concurrency, in this case. Our current code does not even support synchronization primitives other than `join`, so background threads performing imperative operations are not used at all.

```
5   (e0:let () (string:write "e1>\n")
6     (e0:let (next-sexpression) (e0:primitive io:read-sexpression port)
7       (e0:let (results) (repl:macroexpand-transform-and-execute next-sexpression)
8         (e0:let () (repl:write-results results port)
9           (e0:let () (string:write "\n")
10            (repl:loop port)))))))
```

5.4.1.6 An aside: developing, testing, and the ordering of phases

In this presentation we have chosen to show the *final structure* of our bootstrapping system as a working body of code, rather than recounting the *process* of writing it; the two views do not perfectly overlap.

The preceding phase is by far the most problematic in this respect: as any reader with implementation experience may witness ε_1 with its interpreter, global data structures, macro and transform systems is a very strongly recursive system, where each component tends to require all the others in a loop of circular dependencies apparently very difficult to break. And indeed, the preceding third phase was not easy to implement on the machine.

After deciding on the general bootstrapping strategy we wrote a first approximation of the system, in ε_0 with with no macros (see next phase) and no transforms, up to the interpreter included. Some subsystems, for example the implementation of sum-of-products types for ε_0 expressions, were first prototyped in Guile. Transformations were added as the very last step, after macros worked reliably and were used to make ε_1 considerably more friendly.

With the marshalling/unmarshalling support needed for *unexec* (§3.3.2), we followed a route of progressively reducing the abstraction level: after writing its first version in ε_1 using several comfortable language extensions, we translated it into ε_0, to make it possible to run it earlier at boot, when extensions are not loaded yet. The translated marshalling code is understandable, but some complexity which would have been a little too daring for ε_0 still shows up in the code, particularly in nested conditionals.

Later we rewrote the marshalling and unmarshalling support for a third time in C, for performance reasons (§5.4.3).

At the beginning we wrote a considerable body of debugging code in Scheme, including for example for example the procedure `print-expression` writing expressions in ε_0's syntax of §2 including handles in Unicode subscript digits, or `hash-dump-sizes` which has served to test how well our hash functions distribute; and maybe most importantly `meta:print-procedure-definition` and `meta:print-macro-definition`, useful for inspecting the global state and obtain readable syntax. Such code is still available in `bootstrap/scheme/conversion.scm`, and still occasionally useful for debugging:

```
1  guile> (meta:print-procedure-definition 'cons:make)
2  Formals: (car cdr)
```

```
[let [result] be [call buffer:make 2779]780 in [let [] be [call buffer:set! result781 0782
car783]784 in [let [] be [call buffer:set! result785 1786 cdr787]788 in result789]790]791]792

guile> (meta:print-macro-definition 'e0:call)
(sexpression:inject-expression (e0:call* (sexpression:eject-symbol (sexpression:car arguments))
(e1:macroexpand-sexpressions (sexpression:cdr arguments))))

guile> (e0:value whatever:identity) ;; the symbol dump is painful to read
0x1471040[0x14c41e0[17 119 104 97 116 101 118 101 114 58 105 100 101 110 116 105 116 121]
0 127 0x1409f00[0x14633c0[0x145c900[1 120] 0 127 0 0 0 0 0 0] 0] 0x1462b80[0 10 0x14633c0]
0 0 0 0]
```

But as our design of ε_1 changed until its crystallization into the present form, some of our crude debugging and code-generating tools also broke down and became unusable, whenever their underlying assumptions failed. As old scaffoldings not supporting any more a structure now able to stand by itself, we abandoned them.

Our bootstrapping code running on top of an inefficient extension to Guile had low performance, which was unsurprising. What we didn't expect was that waiting times were in practice so unbearable even on our fastest machine[24] that it necessitated optimizations already in this phase. §5.4.3 provides some insights.

5.4.1.7 Phase *(iv)*: fill reflective data structures

Phase *(iii)* consisted of about 2500 lines in ε_0, which we have executed on top of the ε_0 implementation of Phase *(ii)* based on `guile+whatever`; in other words, our global definitions up to this point affected *Guile's state environments*, rather than ours. This phase consists in using the Guile data structures we updated at each definition to fill "reflective data structures" — in quotes, since we are actually speaking of data to be stored as part of symbol objects.

The code is in `bootstrap/scheme/fill-reflective-structures.scm`.

Our Scheme implementation of `e1:define` from Phase *(ii)*, at the end of `bootstrap/scheme/epsilon0-in-scheme.scm`, updates two global Scheme data structures: `globals-to-define`, a list of names of non-procedure globals which have been defined, and `procedures-to-define`, an alist binding each defined procedure name to its formals as a Guile list and its body as a Guile s-expression. The idea is to scan the lists and for each element to copy the corresponding data into our state environments.

- Non-procedures are easy to manage: given a global name as a Guile symbol we simply have to look it up as a Guile global: the value we find, an injected whatever, has to be copied into the appropriate field of the ε_1 symbol corresponding to the Guile symbol.

[24] *optimum* is a Dell Precision T7400 with two quad-core Intel Xeon (EM64T) chips at 3GHz, 8Gb of RAM, heavily customized debian GNU/Linux "unstable".

- Procedures are more involved: for each one `procedures-to-define` contains its name as a Guile symbol, its formals as a Guile symbol list, and its body as a Guile s-expression. Name and formals are easy enough to translate, but for our state environment *we need the body as an ε_0 expression* encoded in the expression data structure we defined in Phase *(iii)*; converting each body into an expression is the main problem of this phase.

At this point we can better justify our rigidly constrained way of writing code in Phase *(iii)*, in which we *only* used ε_0 plus `e1:define`: the s-expression-to-expression translation we need to perform at this point is *non-macro expansion*. Since the translation has to be executed only once without the translation code itself being part of the output, we can implement it in Scheme rather than in ε_0. The procedure `e1:non-macro-expand`, defined in a mutually-recursive fashion with its helpers `e0:non-macro-expand-sexpressions`, `e0:non-macro-expand-symbols` and `e0:non-macro-expand-values`, follows very closely our Definition 5.8.

The code is slightly less readable than the corresponding mathematical definition just because of explicit representation conversions between Guile's and ε_1's data; for example, the procedure `whatever->guile-boolean` converts an untyped ε_1 object into a Guile dynamically-typed boolean, and `guile-sexpression->sexpression` converts a native Guile s-expression into our own representation, as per the previous phase. All such conversion operators, by themselves quite unremarkable, are implemented in Scheme, in `bootstrap/scheme/conversion.scm`.

Our `e0:non-macro-expand` is also "unsafe" and in practice accepts a superset of valid syntax encodings: we avoided safety checks in the code, for example ignoring the s-cddr of `(e0:join x . s)` instead of verifying that it really is `()`. This expansion unsafety is not a problem in practice at this point, since the code to be translated has already been well tested on ε_0's implementation of Phase *(ii)*, using Guile's interactive REPL (actually `guile+whatever`'s), and just a little care in reading untyped data structure dumps (§3.3.1).

The real work is in the Scheme procedure `set-metadata!`, a zero-parameter procedure which consists of two loops, the first scanning the global binding list and adding the definition to symbols, and the second doing the same for procedures after s-expression conversion and non-macro expansion.

Even on our *optimum* machine, when using Guile 1.8, which is faster in this phase, the computation of Phase *(iv)* takes about 15 seconds, compared to 0.2 seconds for the previous phases combined; fortunately, unless there are recent changes in `core.e`, we can in practice entirely skip this phase by *exec*ing (§3.3) over the Phase *(ii)* interpreter.

One problem remains: the globals and procedures we have defined up to this point *also* remain in Guile's state environments, and this state of things will persist up until we remove the dependency on Guile. Re-defining some procedure directly invoked from Guile, would lead to subtle problems, making the two definition sets

inconsistent. We will simply avoid to override any ε_1 definition with an incompatible one.

With the caveat above, and now having global and procedure definitions in place, we can finally use `e1:eval`.

5.4.2 Unexec

At this stage it is finally possible to use unexec, which depended on reflective structures to dump a program. Our vague reference in §3.3.1 to the "surprisingly few" data structures involved should be clear now: a simple way of obtaining a program is to dump a pair containing:

- the *symbol table*, holding all global and procedure definitions and from which all alive data in memory can be reached (§3.3.1);
- the *main expression*.

At *exec* time, it suffices to unmarshal the pair, define the symbol table, and run the interpreter on the expression.

The ε_0 implementation of `unexec:unexec` and `unexec:exec` is in `bootstrap/scheme/unexec.e`; the same file also contains the ε_0 implementation of marshalling and unmarshalling.

5.4.3 Optimizations

In a preliminary version of ε_1, macros were not associated to procedures to be called, but to expressions to be evaluated. The current definition has a cleaner interaction with transforms, but if we ignore transforms the old solution was perfectly workable as well: instead of passing parameters to a procedure, we evaluated an expression in some environment, with the same effect.

With the old solution, macroexpansion returned correct results, but the system's incredible inefficiency led us to investigate the issue until we discovered a perverse pattern: the complicated circular nature of the dependencies between `e1:macroexpand`, expression-expanders, `e0:eval` and its helpers made it difficult to understand how, indirectly, `e0:eval` *was interpreting calls to itself.*

It is easy to see how, if adding one layer of interpretation worsens performance by some constant factor k, we have that a stack of n interpreters has exponential overhead k^n; and given that symbolic interpreters easily cause order-of-magnitude overheads, the slowdown was evident even for very small values of n.

For the first time in our programming experience we discovered that some code had *unbounded interpretation overhead.* Despite being now unnecessary because of the macroexpansion changes, we still find the problem and its solution quite beautiful, and potentially instructive for others.

Implementation Note 5.11 (The Hack) *When evaluating a call to* `e0:eval`*, the self-interpreter does not evaluate* `e0:eval`*'s body, but directly the given expression in the given local environment.* □

The idea is simply to recognize as a particular case any call of a procedure named `e0:eval`:

```
(e1:define (e0:eval-call name actuals local)
  (e0:if-in (whatever:eq? name (e0:value e0:eval)) (#f)
    (e0:eval-non-eval-call name actuals local)
    (e0:eval-eval-call actuals local)))

(e1:define (e0:eval-eval-call actuals local)
  (e0:let (actual-values) (e0:eval-expressions actuals local)
    (e0:if-in (whatever:eq? (list:length actual-values) (e0:value 2)) (#f)
      (e1:error (e0:value "e0:eval-eval-call: in-dimension mismatch"))
      (e0:let (expression) (list:head actual-values)
        (e0:let (local) (list:head (list:tail actual-values))
          (list:singleton (e0:eval expression local)))))))  ; wrap as inner eval would
```

Subjectively, it could be said that *The Hack* changed the interpreter from being *comically* slow to being still very slow, but at least usable.

Despite being an obvious idea, the following implementation aspect deserves prominence because of its dramatic impact on performance:

Implementation Note 5.12 (interpreter in C) *We re-implemented an* ε_0 *interpreter in low-level C, using explicit stacks and no heap allocation, except implicitly for building whatevers. The C implementation is a few hundreds of lines long, and performs runtime dimension checks. The interpreter is available from* ε_1 *as the primitive* `e0:fast-eval`*, and has the same interface of* `e0:eval`*.* □

Replacing the ε_0 self-interpreter with the C implementation led to a speedup of around 200 for an exponential-time recursive implementation of Fibonacci's function:

```
(e1:define (fibo n)
  (e0:if-in n (0 1)
    n
    (fixnum:+ (fibo (fixnum:- n (e0:value 2))))
              (fibo (fixnum:1- n)))))
```

The high speedup is not surprising, if we consider that the ε_0 self-interpreter had to run on top of Guile, itself an interpreter.

The "Interpreter in C" strategy subsumes The Hack: being written in a different language than ε_0, the interpreter never accesses its own body, and interpreter calls in the interpreted code are instead executed by a primitive, thus avoiding overhead multiplication.

Implementation Note 5.13 (*exec*/*unexec* in C) *We re-implemented the marshalling and un-marshalling procedures* `unexec:dump` *and* `unexec:undump` *by using primitives written in C. The C implementation consists of about 100 lines, and adopts exactly the same data structures and algorithm as the corresponding* ε_0 *code.* □

Again, the optimization of Implementation Node 5.13 has an order-of-magnitude impact on performance: thanks to it, *exec* "quick-start" takes only a short fraction of a second on *optimum*.

Re-implementing part of the functionality in C was an aid to development and rapid testing, more than a definite commitment: after a good native compiler is developed, the need for such optimizations will attenuate.

5.4.4 Sample extensions

The file `bootstrap/scheme/toplevel-in-scheme.scm`, run right after `fill-reflective-structures.scm`, defines a few simple Scheme macros to let the user evaluate ε_1 forms within the Guile REPL: "`(e1:toplevel . s)`" evaluates each element of the s-list s as an ε_1 expression, which of course is macroexpanded and transformed before execution. "`(e1:trivial-define-macro` m s`)`", available both as a Scheme macro and as an ε_1 macro, defines the macro named m (an s-symbol) as s (a generic s-expression).

Armed with just this knowledge, the reader should be able to follow quite easily `bootstrap/scheme/epsilon1.scm`, which contains around 2000 lines worth of ε_1 extensions.

We think that the power of syntactic abstraction is easy to appreciate now, by looking at how fast language expressivity improves after each definition, compared to the development work in phase *(iii)* during which only procedural abstraction was available.

This sequence of extensions, quite impressive in its accelerating rhythm, raises the language from the clumsy beginnings of ε_0 to a respectable power, with *sequences*, *multi-way conditionals*, *short-circuit logical operators*, Common Lisp-style *destructuring macros*, *variadic procedures*, *tuples*, *records*, *extensible sum-of-product definitions*, *closures*, *imperative loops* and *futures*.

Interestingly, only the three last extensions in the sequence above depend on a transform; macros alone can already bring the language to a quite high level.

Most of our syntactic conventions are inspired to Scheme, and the form names are indeed largely compatible, apart from the "`e1:`" prefix.

The very beginning of `bootstrap/scheme/epsilon1.scm` deals with macros for core ε_0 forms:

```
;;; These first crude versions do not perform error-checking, silently
;;; ignoring additional subforms at the end.
(e1:trivial-define-macro e0:variable
  (sexpression:inject-expression
    (e0:variable* (sexpression:eject-symbol (sexpression:car arguments)))))
(e1:trivial-define-macro e0:let
  (sexpression:inject-expression
    (e0:let* (sexpression:eject-symbols (sexpression:car arguments))
             (e1:macroexpand (sexpression:cadr arguments))
             (e1:macroexpand (sexpression:caddr arguments)))))
```

The code is simple, but it is questionable whether it really belongs in this file, rather than in `core.e`. The reason why we defined such important features so late is mostly pragmatic: such macro definitions would have been be much less comfortable to write using only `state:macro-set!` without `e1:trivial-define-macro`. For similar reasons we defined quoting and quasiquoting in this file, rather than in `core.e`.

The debate about where exactly the ε_1 "core" ends and "extensions" begin looks futile anyway, and indeed the very notion of personality, possibly useful for humans to identify a set of features, has no consequence for the implementation. The same objection may be raised "at the other end", about the CPS transformation and the reason why we defined in its own source file instead of in the end of `epsilon1.scm`. In the same somewhat arbitrary fashion, we proclaim that continuations do not belong in ε_1 but are part of another experimental personality *based on* ε_1. First-class continuations provide another qualitative jump in expressivity, but our implementation is less mature and quite expensive in terms of performance, therefore less appropriate as part of the general-purpose "library" to build personalities which ε_1 is meant to be.

In the following we will just add some quick considerations about the main extensions.

5.4.4.1 Quoting and quasiquoting

Quoting and *quasiquoting*, heavily relying on the type table (§5.4.1.3) so that support for new new types can be added smoothly, are different from their Lisp homologous: in ε_1 *a quoted or quasiquoted s-expression yields an expression which will build that s-expression when evaluated*; for example `'1` macroexpands to a procedure call of `sexpression:inject-fixnum` which, if evaluated, will build *the s-expression* (not the unboxed fixnum) `1`.

Despite this difference, we can easily adapt the standard algorithms for quasiquoting[25], which is convenient since *nested quasiquoting* is famously tricky to implement

[25] We followed Bawden's updated proposal (different from his older one in [7, §B]), as quoted by Kent Dybvig at `http://www.r6rs.org/r6rs-editors/2006-June/001376.html`. This new version was eventually adopted in [79].

correctly.

The non-homoiconicity of ε_1 forces us to think of the difference between s-expressions, uninjected values and expressions, and costs us some injection and ejection operators in macros. The inconvenience in practice is tolerable, and we consider the advantages of our syntactic extensions well worth this minor trouble.

5.4.4.2 Variadic procedure wrappers

All practical Lisps permit to define *variadic* procedures, which is to say procedures with an arbitrary number of optional arguments. ε_0 and ε_1 *do not*, for reasons of efficiency. Anyway we can recover the convenience of variadic calls by introducing *variadic macros*, and using them to wrap procedures.

The following ε_1 definitions extend binary operators with a neutral element to make them accept any number of arguments:

```
1 (variadic:define-associative fixnum:+ fixnum:+ 0)
2 (variadic:define-right-deep fixnum:** fixnum:** 1)
```

The macro-call expansions of `variadic:define-associative` and `variadic:define-associative` *generate more macro definitions*, in this case for "`fixnum:+`" and "`fixnum:**`", which for added convenience are also the names of the corresponding procedures.

After the definition, using the debugging procedure `meta:macroexpand` we can examine how variadic calls are always "eliminated" at macroexpansion time, yielding efficient residual code:

```
1  guile> (meta:macroexpand '(fixnum:+)) ;; no arguments: neutral element as a literal
2  0_72531
3  guile> (meta:macroexpand '(fixnum:+ 7)) ;; one argument: no calls are needed
4  7_72532
5  guile> (meta:macroexpand '(fixnum:+ 1 2)) ;; one sum
6  [call fixnum:+ 1_72533 2_72534]_72535
7  guile> (meta:macroexpand '(fixnum:+ 1 2 3 4)) ;; three sums, left-deep (currently faster)
8  [call fixnum:+ [call fixnum:+ [call fixnum:+ 1_72536 2_72537]_72538 3_72539]_72540 4_72541]_72542
9  guile> (meta:macroexpand '(fixnum:** 2 3 4 5)) ;; three calls, right-deep as requested
10 [call fixnum:** 2_72605 [call fixnum:** 3_72606 [call fixnum:** 4_72607 5_72608]_72609]_72610]_72611
```

Since variadic syntax is so convenient, we use it also for of many other macros which are not procedure wrappers:

```
1 guile> (meta:macroexpand '(e1:or))
2 0_72464
3 guile> (meta:macroexpand '(e1:or a))
4 a_72465
5 guile> (meta:macroexpand '(e1:or a b c))
6 [if a_72466 ∈ {0} then [if b_72467 ∈ {0} then c_72468 else 1_72469]_72470 else 1_72471]_72472
```

```
guile> (meta:macroexpand '(e1:and a b c))
[if a72473 ∈ {0} then 072474 else [if b72475 ∈ {0} then 072476 else c72477]72478]72479
```

5.4.4.3 Sum-of-product types

Sum-of-product or *sum* types are a kind of variant records, introduced by ML and popular in the functional programming community.

Even in ε_1's untyped context, it is very convenient to automatically turn a sum description into procedures for building, accessing and updating data, and for testing the case of a given object.

As a classic example, let a list be either nil, or the cons of a head and a tail:

```
e1> (sum:define my-list (nil) (cons head tail))
Defining the procedure my-list-nil...
Defining the procedure my-list-nil?...
Defining the procedure my-list-nil-explode...
Defining the procedure my-list-cons...
Defining the procedure my-list-cons-make-uninitialized...
Defining the procedure my-list-cons-explode...
Defining the procedure my-list-cons-get-head...
Defining the procedure my-list-cons-with-head...
Defining the procedure my-list-cons-set-head!...
Defining the procedure my-list-cons-get-tail...
Defining the procedure my-list-cons-with-tail...
Defining the procedure my-list-cons-set-tail!...
Defining the procedure my-list-cons?...
```

Our sum type definitions keep into account the number of cases which must be represented as boxed, and do not generate tag fields unless needed. We derive the representation in memory from the sum definition in a way similar to [5, §4.1].

```
e1> (my-list-nil) ;; unboxed
0
e1> (my-list-cons 10 (my-list-nil)) ;; just head and tail, no case tag
0x1b984d0[10 0]
e1> (my-list-cons 10 (my-list-nil)) ;; make a *new* cons: different address
0x1be3b70[10 0]
e1> (my-list-cons 10 20) ;; "ill-typed" as a list: the system doesn't care
0x1998350[10 20]
e1> (my-list-cons? (my-list-nil)) ;; is nil a cons?  (No, it's not)
0
e1> (sum:define complex (cartesian real imaginary)
                        (polar angle radius)) ;; two boxed cases: case tag needed
;;; [...]
e1> (complex-cartesian 100 200) ;; first case: tag 0
0x152c0c0[0 100 200]
e1> (complex-polar 100 200) ;; second case: tag 1
0x1502620[1 100 200]
```

We can now redefine ε_0 expressions as an *open* sum-of-products, openness meaning that more cases can be added later. This permits more flexibility, at the cost of a slightly less efficient representation in the general case:

```
(sum:define-open e0:expression
  (variable handle name)
  (value handle content)
  (bundle handle items)
  (primitive handle name actuals)
  (let handle bound-variables bound-expression body)
  (call handle procedure-name actuals)
  (call-indirect handle procedure-expression actuals)
  (if-in handle discriminand values then-branch else-branch)
  (fork handle procedure-name actuals)
  (join handle future)))
```

The representation is compatible with the one used in `core.e`, and from now on it will also be possible to add new expression cases, for user-defined expression forms.

5.4.4.4 Closure Conversion

The purpose of this extension is adding *statically-scoped*, *higher-order anonymous procedures* to ε_1, implemented as *closures*.

Anonymous procedures require two[26] new syntax cases, `lambda` and `call-closure`:

```
(sum:extend-open e0:expression
  (lambda handle formals body)
  (call-closure handle closure-expression actuals))
(e1:define (e1:lambda* formals body) ;; make a lambda expression
  (e0:expression-lambda (e0:fresh-handle) formals body))
;;; "Concrete syntax" for lambda, generating the new expression case.  This is a
;;; variadic macro of one or more arguments: body-forms is bound to the argument s-cdr.
(e1:define-macro (e1:lambda formals . body-forms)
  (sexpression:inject-expression
    (e1:lambda* (sexpression:eject-symbols formals)
                (e1:macroexpand `(e1:begin ,@body-forms)))))
```

Our closures are flat and minimal [24, p. 132], consisting of a single buffer holding a procedure name as its first element, followed by its zero or more nonlocal values; for us, nonlocals are the variables *locally-bound out of the lambda-expression, occurring free in the lambda body, hence in particular not shadowed by the lambda formals*. The procedure referred by the closure takes *the closure itself* as a parameter, followed

[26] `Call-closure` does not technically need to be added as a new syntactic case, as it would also be definable as a macro; only the `lambda` case has an expansion which depends on its context. However having both cases representable as expressions is useful for the CPS transform, and may be a good idea in case we want to add type analyses in the future.

by the ones explicitly mentioned as formals, and locally binds the nonlocal names by loading their values off the closure.

For example `(e0:let (a) 57 (e1:lambda (x) a))` will yield a closure of two elements: a procedure name (automatically generated, two parameters: the closure and `x`), and the only nonlocal value `57`. The procedure body will contain an `e0:let` block binding the name `a` to the second element of the buffer pointed by its closure parameter.

Calling a closure is easy: given a closure and its actuals, we load the first element referred by the closure and we perform an indirect call to it, passing as parameters the closure itself followed by the actuals.

`closure:closure-convert` implements closure-conversion; it needs a procedure and the set of locally-bound variables. The procedure is very simple, based on a multi-way conditional `e1:cond` which dispatches on the expression case[27]. `e1:let*`, not to be confused with `e0:let*`[28], is a block binding sequentially:

```
1  (e1:define (closure:closure-convert e bounds)
2    (e1:cond ((e0:expression-variable? e)
3              (e0:variable* (e0:expression-variable-get-name e)))
4             ((e0:expression-bundle? e)
5              (e0:bundle* (closure:closure-convert-expressions
6                           (e0:expression-bundle-get-items e) bounds)))
7             ;; [...] other trivial cases [...]
8             ((e0:expression-lambda? e) ;; Interesting case
9              (e1:let* ((formals (e0:expression-lambda-get-formals e))
10                       (nonlocals (set-as-list:subtraction bounds formals))
11                       (old-body (e0:expression-lambda-get-body e))
12                       (new-body (closure:closure-convert old-body
13                                                          (set-as-list:union bounds formals)))
14                       (used-nonlocals (set-as-list:intersection nonlocals
15                                                                 (e0:free-variables new-body))))
16                ;; closure:make* defines a global procedure, then returns an expression
17                ;; which builds a closure data structure including the global procedure name
18                (closure:make* used-nonlocals
19                               (closure:variables* used-nonlocals)
20                               formals
21                               new-body)))
22            ((e0:expression-call-closure? e) ;; The second interesting case
23             (e1:let* ((closure-expression (e0:expression-call-closure-get-closure-expression e))
24                       (actuals (e0:expression-call-closure-get-actuals e))
25                       (transformed-closure-name (symbol:fresh)))
26               (e0:let* (list:singleton transformed-closure-name)
27                        (closure:closure-convert closure-expression bounds)
28                        (e0:call-indirect*
```

[27] *Pattern-matching* over sum types can be implemented with macros, and in fact we did that in a previous prototype: see §5.4.5.

[28] The naming convention is unfortunate in this case, but the sequential-binding name "`let*`", as distinct from parallel-binding "`let`", is convenient and has been conventional in Lisp for decades.

```
                            (e0:primitive* (e0:value buffer:get)
                                           (list:list (e0:variable* transformed-closure-name)
                                                      (e0:value* 0)))
                            (list:cons (e0:variable* transformed-closure-name)
                                       (closure:closure-convert-expressions actuals bounds))))))
           (else
            (e1:error "unknown extended or invalid expression"))))
(e1:define (closure:closure-convert-expressions es bounds)
  (e1:if (list:null? es)
    list:nil
    (list:cons (closure:closure-convert (list:head es) bounds)
               (closure:closure-convert-expressions (list:tail es) bounds))))
```

`closure:closure-convert` is the basis of our procedure transforms:

```
(e1:define (closure:closure-convert-expression-transform expression)
  (closure:closure-convert expression set-as-list:empty))
(e1:define (closure:closure-convert-procedure-transform name formals body)
  (e0:bundle name
             formals
             (closure:closure-convert body formals)))

(transform:prepend-expression-transform! (e0:value closure:closure-convert-expression-transform))
(transform:prepend-procedure-transform! (e0:value closure:closure-convert-procedure-transform))
```

Now that we have installed the transform procedures, we can use closures:

```
e1> (e1:define q (e1:let* ((a 1) (b 2) (c 3))
                   (e1:lambda (x)
                     (fixnum:+ a b c x))))
e1> (e1:call-closure q 4)
10
```

It should be remarked that closures are distinct from and incompatible with ε_0 procedures. Should we hide ordinary procedures from the user, and use closures only?

We could: it is possible to introduce (trivial) closures for all existing procedures, retroactively transform away all procedure calls into closure calls (and then into indirect calls by closure-conversion) and finally change the cons-expander to generate a *closure call* rather than a procedure call as its default case. This would make ε_1 similar to a "Lisp-1" [32] by hiding from the user the existence of procedures which are independent from closures.

Such a move would be perfectly reasonable in many high-level personalities, but we reject it for ε_1, for which we want to retain low-level control.

5.4.4.5 Futures

Our `fork` form in ε_0 is very inconvenient to use, needing a procedure which must be given parameters to evaluate in foreground, rather than just an expression (§2.2, p. 22). But closures make it easy to define friendlier futures, by a simple macro:

```
(e1:define (future:fork-procedure thread-name future-closure)
  (e1:call-closure future-closure))

;;; Build a future which will asynchronously call the given closure:
(e1:define (future:asynchronously-call-closure closure)
  (e0:fork future:fork-procedure closure))

(e1:define-macro (e1:future . forms) ;; friendly syntax: any number of forms in sequence
  `(future:asynchronously-call-closure (e1:lambda () ,@forms)))
```

5.4.4.6 First-class continuations

We implemented first-class continuations with a CPS transform [83, 5, 44, 46] on expressions extended with a `let/cc` form ("`CATCH`" in [89]), with `call/cc` defined as a macro over `let/cc`.

Our CPS transform is more tentative than the ε_1 personality, and currently resides in `bootstrap/scheme/cps.scm`, and the trivial driver `bootstrap/scheme/cps-repl.scm`. Implemented in a very conventional style, it works but yields inefficient code and is inefficient at transformation time as well: in particular the high number of local variables generated by CPS stresses closure-conversion and its algorithm to compute the free variables of an expression, currently quadratic.

The generated code allocates closures at a very high rate; it can be optimized and some improvements appear easy, but to obtain really efficient code we would need escape analysis, so that code sure not to escape could be recognized and transformed differently. Such global (or just "incremental") analyses can be performed in our model, by having a CPS transform return its provisional inefficient result but save the original untransformed code, to be reconsidered later.

Bundles have been problematic, since CPS maps our `e0:let` form, which ignores excess items (§2.1.5, p. 20), into a procedure call, which does *not* ignore excess parameters; in order to respect our `e0:let` semantics we had to relax some dynamic checks in the ε_0 interpreter, and rely on a behavior which constitutes an error according to the semantics. It is not clear whether it would be best to update the semantics to ignore extra parameters (hence *defining a non-error behavior in more cases*, which constrains implementations[29]), or to forbid bundles altogether in conjunction with CPS.

Continuations have been very useful to test and stress our transform system, since a CPS transform is much less "well-behaved" than a closure-conversion transform: CPS adds one more argument to *every* procedure, making transformed code fundamentally incompatible with its original version. When closures are not used,

[29] We only touched the C version, by trivially removing two conditionals, one for `e0:call` and the other for `e0:call-indirect`. The same change can be easily replicated in the ε_0 self-interpreter. In such symbolic interpreters removing the check is trivial and actually slightly *improves* performance: this will not be true in a compiled implementation.

closure-conversion returns unchanged code, up to handles; but a (naïve) CPS transform fundamentally changes the expression structure even where no jump is performed. Of course the CPS transform needs to be applied *retroactively* (§5.4.1.5, p. 106).

We are not positive about traditional "full" continuations being pragmatically the best foundation to base further extensions on; *delimited continuations* [33, 68, 27] seem to provide some advantages, and we have experimented with them in early prototypes; thanks to our open-ended design we may adopt them in the future.

5.4.5 Implementation status

The implementation is not mature, but it can be played with. We currently depend on Guile to parse and print s-expressions, and our current implementation still lacks a compiler.

Such limitations are temporary and incidental: in the time available we chose to develop transforms, more innovative and interesting, rather than implementing well-known algorithms once again. We do not envisage any particular difficulty, and development will proceed during the following months.

Some older prototypes, unmaintained but available at `http://ageinghacker.net/epsilon-thesis-prototypes/`, contain code which could possibly be worth adapting and integrating into the current implementation:

- an s-expression frontend written in OCaml for an older prototype, supporting the grammar of Figure §5.1; it works and contains a very powerful scanner supporting a variant of Thompson [91] and Rabin-Scott Constructions [69] over large character-sets;
- an incomplete compiler including liveness analysis and RTL generation;
- pattern-matching macros working on a different implementation of sum-of-product types;
- a mostly complete CamlP4 printer, intended to automatically translate the OCaml code into maintainable ε_1 code.

An official part of the GNU project, epsilon is free software, released under the GNU GPL version 3 or later [31]. Its home page is `http://www.gnu.org/software/epsilon`.

We manage the source code on a public bzr server at `bzr://bzr.savannah.gnu.org/epsilon/trunk` [2015 note: switched from bzr to git in late 2013: see `https://savannah.gnu.org/git/?group=epsilon`], and a public mailing list is available for discussion. See `https://savannah.gnu.org/projects/epsilon` for more information.

5.5 Future work

Building a large body of extensions raises the issue of controlling their interaction. Transform-based extensions in particular, relying as they do on the enumeration of all expression forms, require knowledge of all the previously-added expression forms. No solution to this problem is apparent. However, without promising a "silver bullet" to language extension, we still maintain our approach of *layered* syntactic forms to be much more suitable to extensibility than the traditional solution of a large unstructured collection of language forms.

As an orthogonal problem, our current implementation does not currently keep a map from expressions to original source locations (§1.3.4), which may complicate debugging. Ad-hoc solutions involving an s-expression frontend keeping track of source locations, then to be threaded through macros and transforms up to the final generated code, seem perfectly feasible, with handles coming in handy; on the other hand it is desirable to keep extension definitions as uncluttered as possible, ideally by leaving the "current" location information always implicit at each stage, in a monadic fashion. A clean solution to this problem seems well worth investigating.

5.6 Summary

Lisp is a powerful language, and its homoiconic syntax based on s-expressions makes it easy to extend with macros.

We adopted a form of Lisp-style s-expressions as a data structure to represent user syntax, but we keep it distinct from expressions: our macros map s-expressions into expression objects; then, going beyond Lisp, expression objects can be manipulated by user-specified transform procedures, until all syntactic extensions are "transformed away" and only ε_0 forms remain.

We have shown in detail how ε_1, a low-level ε personality useful as a basis to build other extensions, is bootstrapped from ε_0 temporarily leaning on Guile. Our bootstrapping code also constitutes a complete definition of the macro and transform systems.

We closed by showing some interesting language extensions in ε_1, as representative examples of our syntactic abstraction facilities.

Chapter 6

A parallel BiBOP garbage collector

When a high-level program requiring garbage collection runs in parallel on a multi-core machine, the memory subsystem easily becomes the bottleneck. For this reason we implemented a parallel collector for ε, actually starting back when the current incarnation of the language was still taking shape, testing it on a toy Lisp implementation we originally wrote as a teaching aid.

The collector's performance profile is meant to best match a mostly-functional personality. It is relatively easy to interface to C systems, and by design is not limited to ε.

We call our system "`epsilongc`". As for the language name, the initial "`e`" is always written lowercase.

Contents

Our parallel collector is non-moving, based on a variant of the BiBOP organization. Building on the experience of Boehm's work and revisiting some older ideas in the light of current hardware performance trends, we propose a design leading to compact data representation and some measurable speedups, particularly in the context of functional programs.

This effort results in a clean architecture based on just a few data structures, which lends itself to experimentation with alternative techniques.

6.1 Motivation

In recent years improvements in processor performance have been due more and more to increased parallelism, while the trend of rising processor clock frequency has dramatically slowed down. In contrast to what happens with instruction-level parallelism, the task parallelism offered by modern multi-cores must be explicitly exploited by the software, if *any* speedup is to be obtained [90].

As multi-core architectures support a shared-memory model[1] the techniques presented here extend from the now ubiquitous desktop multi-core machines to the older multi-socket SMPs, and to most recent medium-size parallel machines containing several multi-core CPU dies.

The architecture we illustrate here is also suitable for sequential machine, but the need for such a software is particularly stringent in a parallel context. In a sense the rise of the number of CPUs *amplifies* the memory wall problem: the memory bandwidth is a limited hardware resource which all cores have to share, and raising the parallelism degree inevitably tightens up the bottleneck, even without any synchronization.

6.1.1 Boehm's garbage collector

Boehm's garbage collector [15, 12] is the natural point of comparison for our work because of several design similarities, including the idea of (partially) conservative pointer finding, and the use of Unix signals to interrupt mutators[2]. For this reason it may be worth to quickly highlight the main objectives we have set forth for our implementation, in order to better explain the need for our effort and to illustrate key similarities and differences. Our objectives also more or less dictate several design and implementation choices which we prefer to make explicit from the beginning.

First of all, C is clearly the language providing the best control on performance for such a low level implementation where each memory access matters. A slightly less obvious choice is determined by the typical usage of *parallel* systems, tending to concentrate on bulk processing rather than interactive applications: for this reason we consider *bandwidth*, and not latency, to be a priority; this choice excludes most incremental schemes and favors a *stop-the-world* model where many threads can mutate in parallel or collect in parallel, but without any time overlap between the two phases — all of which is similar to Boehm's solution. Since we are interested in the allocation pattern of functional programs, consisting in a large number of small objects, it is paramount to make a good use of the limited space in the primary and

[1] The architecture shown here does not generalize so well to NUMA machines, more suitable as they are to a message-passing style where each task runs in its own addressing space; message-passing is also interesting, as the same interfaces could scale up to parallel computation *over the network*.
Moving away from thread parallelism to pure *process* parallelism (one heap per process) would essentially eliminate the problem of parallel non-distributed garbage collection, but such a revolution appears unlikely. Other organizations like NUMA machines composed by SMP nodes, or machines where the NUMA effect is pronounced only between "distant" nodes, look more realistic and are already being adopted by some current high-class machines [23]. For such a hybrid SMP-in-NUMA model the techniques shown here apply at the SMP level, just in the same way as they would apply to each single machine in a cluster of SMPs.

[2] Notwithstanding the outdated information at `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcdescr.html` Boehm's collector now also employs signals to stop mutators on all major platforms except Windows, where Unix signals are not supported but an analogous mechanism exists for suspending a thread from another thread. [13] mentions GNU/Linux, Solaris, Irix and Tru64. The Windows implementation in `win32_threads.c` uses signal-like primitives like `SuspendThread()`.

secondary caches (henceforth simply *L1* and *L2*), by tightly packing objects together: we want to avoid padding space between heap objects and not to force alignment constraints not specified by the user. Anyway, even if functional programs are our first concern, we would like our collector to be also useful for (human-written) C programs, which encourages us to adopt a non-moving strategy like *mark-sweep* and to avoid safe-points and use *conservative pointer finding* **for roots**; on the other hand there is no reason why other heap objects should not be traced exactly. The collector API should be usable by humans, but not necessarily similar to `malloc()` — an important difference with respect to Boehm's collector.

6.1.2 High-level design

Most of our implementation ideas rely on a variant of the classic BiBOP strategy [82, 26] which, despite its simplicity, has been exploited surprisingly little: the only discussion of an actually implemented similar solution that we have found is in a little-cited 1993 paper by E. Ulrich Kriegel, [45].

In the different context of today, we propose BiBOP as a good match for modern multiprocessor architectures.

We cannot claim novelty for most ideas, some of which are variations of very old implementation techniques, as it is understandable after fifty years[3] of research.

Nonetheless, we feel that our organization may have at least some aesthetic value, in terms of its data structures and C interface.

Our main idea is that *the BiBOP scheme is appropriate for reducing memory pressure on machines with modern memory hierarchies*; we describe this point by introducing the concept of *data density* which we show to be at least one reason for the good performance of our implementation.

6.1.3 The functional hypothesis

Functional programs tend to allocate many small objects, the great majority of which have one of only a few possible "shapes"; in practice, most heap objects will be conses, nodes of balanced binary trees, or more generally components of inductive data structures with fixed size and layout, often containing some constant attributes which must be frequently inspected at runtime, such as the tags of our sum types (§5.4.4.3). Depending on the programming style closures might also be allocated in quantity; allocating other objects tends to be statistically much less frequent, hence less critical for performance.
We define the above set of assumptions as the *functional hypothesis*: our system is designed to run most efficiently when such hypothesis is verified, yet `epsilongc` can and does work with any language, and may even be directly employed for user-written C programs.

[3] We remark one last time how McCarthy *also* introduced garbage collection, in his wonderful [51].

6.2 The user view: kinds, sources and pumps

At a very high level, any automatic memory management system serves to provide *an illusion of infinity*: an unlimited stream of objects created on demand, each satisfying some specified requirements such as size and alignment.

Objects which are not useful any longer can be simply ignored: there is no need, in general, for a user interface to the recycling system itself as the whole point of garbage collection is to make object reusing *invisible* to the user, who just keeps creating more objects as if the memory were unlimited.

The user-level API is built upon three main data structures: the *kind*, each instance of which defines one particular set of requirements for a group of homogeneous objects, the *source*, which arranges for the creation of objects of one specified kind, and the *pump*, providing a single mutator thread with objects from a given source on demand, one object at a time.

6.2.1 Kinds

We define a *kind* as the specific representation of a group of homogeneous heap objects. Each kind is characterized by a given *object size*, *object alignment*, a *tracer* function specifying how to mark the pointers contained in an object given its address, and particular *metadata* values: metadata include[4] an integer *tag* and a *pointer*, sharing the same values for all the objects of the same kind. Given a pointer to a heap object, mutators are permitted to inspect, but not modify, its metadata.

In general a kind should not be confused with a *type*: rather than a type it identifies one *case* among the potentially many variants which, together, make up a type. For example a *cons* kind could be defined, but **not** a *list* kind, which would also comprise the empty list case, having of course a different representation — by the way, reasonably unboxed, as in §5.4.4.3.

The tag could be usefully employed in a dynamically-typed language such as Lisp, for example in order to test at runtime whether a given object is, effectively, a cons. In a statically-typed language like ML the tag can encode the constructor of tagged-sum objects. The pointer metadatum can be useful to refer any reflection-related data not fitting in a single integer.

All the needed kinds are typically defined at initialization time, as global structures shared by all mutator threads.

6.2.2 Sources

From the user's point of view a *source* can be seen as a global inexhaustible source of objects of a given kind. In the typical case the user will define exactly one source per kind at initialization time, as an object shared by all mutator threads; after initialization mutator threads will only refer sources to create their pumps.

[4] Even if they currently comprise only tag and pointer, more metadata can be easily added in the future if the need arises.

6.2.3 Pumps

A *pump* is a *thread-local* data structure implementing but one user-level functionality, the creation of an object.

Each mutator thread will create its own pumps referring the shared, global sources, then use its pumps to obtain new objects. Pumps have to be explicitly destroyed at thread exit time.

6.2.4 Kindless objects

The strategy outlined above — creating objects of some kind which has been defined in advance — suffices the great majority of the objects ever created at runtime: for example in Lisp most heap-allocated objects will be (s-)conses, and Prolog heaps will mostly be made of terms. We call *kinded* all the objects created as shown above.

Some other heap-allocated objects do not fit so well in the picture as it is not possible to foresee in advance their exact size: arrays and character strings come to mind[5]. We provide more "traditional" allocation primitives for such *kindless* objects.

Notice how the kindless object API (see Figure 6.1) provides for less control: vector elements can be either *all* potential pointers, or they can be guaranteed by the user to include *no* pointers. There is not much control on metadata either: all objects share the same[6] tag and metadatum pointer; a user requiring more expressive metadata has to explicitly encode them in the payload. For reasons of general applicability and performance, *we assume* ***not*** *to have boxedness tags available (§3.3.2.1).*

6.2.5 Miscellaneous user functionalities:

Other primitives are provided to initialize and finalize the collector, to register and unregister roots, to notify the memory system about new threads or exited threads, to explicitly force a collection, and to temporarily disable collections and re-enable them.

As all of this is canonical and not particularly interesting, we will not further pursue such details.

6.3 Implementation

Despite their visual intuitiveness, the data structures above were designed primarily for efficiency, and the actual role of each structure is not apparent to the user: in particular the central data structure, the *page*, is completely hidden.

[5]Other slightly less obvious cases are *procedure activation records*, which some runtimes of Scheme, Prolog and SML allocate on the heap; if the language supports dynamic code generation even *code blocks* (either machine language or bytecode) might be heap-allocated and garbage collected.

[6]The actual values can be specified at initialization time, but nonetheless they must be the same for all kindless objects; it is typically reasonable to choose some values not used for kinds, so that at least kindless objects can be distinguished from kinded ones.

```
1  /* A tracer is a pointer to a function taking a pointer as its parameter and
2    returning nothing: */
3  typedef void (*epsilongc_tracer_t)(epsilongc_word_t);
4
5  /* Create a kind: */
6  epsilongc_kind_t  epsilongc_make_kind(const size_t object_size_in_words,
7                                        const epsilongc_unsigned_integer_t
8                                        pointers_per_object_in_the_worst_case,
9                                        const size_t object_alignment_in_words,
10                                       const epsilongc_metadatum_tag_t tag,
11                                       const epsilongc_metadatum_pointer_t pointer,
12                                       const epsilongc_tracer_t tracer);
13
14 /* Create a source from a kind: */
15 epsilongc_source_t epsilongc_make_source(epsilongc_kind_t k);
16
17 /* Initialize a (thread-local) pump from a source: */
18 void epsilongc_initialize_pump(epsilongc_pump_t pump,
19                                epsilongc_source_t source);
20
21 /* Finalize a pump before exiting the thread: */
22 void epsilongc_finalize_pump(epsilongc_pump_t pump);
23
24 /* Allocate a kinded object from a thread-local pump: */
25 epsilongc_word_t epsilongc_allocate_from(epsilongc_pump_t pump);
26
27 /* Lookup metadata: */
28 epsilongc_tag_t epsilongc_object_to_tag(const epsilongc_word_t object);
29
30 epsilongc_metadatum_pointer_t
31 epsilongc_object_to_metadatum_pointer(const epsilongc_word_t object);
32
33 epsilongc_integer_t epsilongc_object_to_size_in_words(const epsilongc_word_t object);
34
35 /* Allocate kindless objects: */
36 epsilongc_word_t epsilongc_allocate_words_conservative(const epsilongc_integer_t size_in_words);
37 epsilongc_word_t epsilongc_allocate_words_leaf(const epsilongc_integer_t size_in_words);
38 epsilongc_word_t epsilongc_allocate_bytes_conservative(const epsilongc_integer_t size_in_bytes);
39 epsilongc_word_t epsilongc_allocate_bytes_leaf(const epsilongc_integer_t size_in_bytes);
```

Figure 6.1: `epsilongc`'s essential user-level API.
The source above is directly copied from header files, with only GCC function attributes (to force inlining and such) removed and comments eliminated. Despite looking unconventional the interface is not particularly complex, and in fact is conceived so that performance-critical operations such as `epsilongc_allocate_from()` and metadata lookup functions can be easily re-implemented in assembly, to be generated by a compiler as intrinsics.

Pointers are essential in the implementation of any language requiring dynamic memory allocation, and in order to make pointers easier to recognize at runtime in the absence of boxedness tags and their dereference more efficient[7], we restrict the set of heap pointers considered valid to *word-aligned* pointers; one word is also the minimum size of a heap object representable without space overhead, and all the integers internally used in the implementation are of type `intptr_t`, so that the size of all memory structures remains a multiple of a word size.

The description below will proceed *from the bottom up*: since many data structures and operations are usable with different collection strategies requiring little or no modifications, we illustrate the various possible operations before our way of combining them, in the spirit of separating policy from mechanism.

6.3.1 Kinded objects

We represent each kinded object as a buffer of words, with *no header*; the rationale of this choice is discussed in more depth in §6.3.8, but the main idea is simply to have long packed arrays of objects in memory, without any padding unless absolutely necessary[8].

6.3.2 BiBOP pages

All kinded objects are allocated from data structures called *pages*[9], similar to Kriegel's "STSS cards" [45]: whenever a pump returns a pointer to a new object, the resulting address will refer a word contained in a page.

Each page can only contains objects of *one* kind. For each kind any number of pages, including zero, may exist at any given time.

All pages have the same size, which must be a power of two; the page size is also equal to its *alignment*: the rightmost log_2`epsilongc_PAGE_SIZE_IN_BYTES` bits of a page pointer are always guaranteed to be zero.
A page is divided into *page header*, *mark array* and *object slot array*.

[7] On many RISC architectures pointers to misaligned objects may not be just a performance concern: some processor families such as *MIPS* and *Sparc* simply raise an exception in response to any attempt to dereference a non-word-aligned pointer. Others, such as the *x86* family, execute the misaligned dereference, but imposing a heavy execution time penalty.

We prefer to simply forbid such pointers for all architectures, which may improve performance and helps to avoid the misidentification of many false pointers.

We also assume convertibility from integer to pointer and vice versa without loss of information: even if not mandated by the C Standard (the type `intptr_t` itself is optional in [37]) such an assumption is in practice true on all architectures.

[8] Padding *must* to introduced sometimes in order to respect the alignment constraints stated by the user: for example the user might require a three-word structure to be aligned to two or four words; in such cases there is no way to avoid wasting some space for each object.

[9] There is no *a priori* relation between BiBOP pages and operating system pages, whose sizes may well be different: BiBOP pages will typically be at least a few times larger than operating system pages, but still smaller than the L2 cache. In the following we always use the term *page* to mean "BiBOP page".

Page header The page header contains a copy of the kind metadata, which of course are valid for all the objects in the page; the object referred by the metadatum pointer, if any, is shared by all the pages of the same kind: only the pointer is copied.

Other information contained in the header includes kind-dependant data such as the object size and effective size, the payload offset, and the number of object slots in the page. All of this is computed once and for all when a kind is created, and simply copied at page initialization time. The address of the first dead slot (see below) is also held in the header.

Since the header has offset zero within the page, given a pointer to any kinded object, *even interior*, the address of its page header can be trivially obtained by *bitwise and*ing the pointer and the *page mask*, defined as the *bitwise negation* of `epsilongc_PAGE_SIZE_IN_BYTES` -1. This allows mutators to access metadata at runtime with an overhead of two to four assembly instructions, when needed; on the other hand the negligible space overhead of storing metadata once per page makes this solution completely acceptable even for languages which don't make use of them.

Mark array The mark array is placed right after the header, with no padding: since the header size is a multiple of the word size, the mark array is guaranteed to always begin at a word boundary.

The mark array stores liveness information for each object[10] in the page: since we currently need only one bit per object, the array could conceptually always be implemented as a bit vector.

As marking is parallel, mark arrays are concurrently updated by several threads, which requires some atomic memory accesses (see §6.3.6). On many machines byte stores are always atomic, and even when suitable atomic instructions for bitwise operations are provided working with a *byte vector* may be more efficient[11]. On (hypothetical) architectures where the compiler did not support the required intrinsics, and where an atomic byte store were not provided, one could use a *word vector*. The implementation allows the user to choose at configuration time among bit, byte or word, bit being the default.

Alternatively, it is possible to enable *out-of-page mark arrays* at configuration time, so that mark arrays are stored as separate `malloc()`ed buffers; in this case the mark array area in a page degenerates to a single pointer, and accessing the mark array from a page requires one indirection. Our original rationale for implementing this strategy was to avoid some cache conflict misses due to the fact that mark arrays

[10] It is interesting to compare this with Boehm's collector, which stores one element per object *word*, thus making tracing simpler. We have chosen to slightly complicate the mapping from mark array elements to objects instead, to speed up the critical operation of page sweeping, and in general trading more computation for fewer memory accesses.

[11] [12], written in 2000, compares the solutions on several architectures, finding that the optimal solution depends on the machine. According to our recent tests, the best strategy between bit arrays and byte arrays remains machine-dependent.

share the same alignment on all pages. As benchmarks showed that this is not a problem in practice with modern multi-way set associative caches, this strategy has not been pursued further by separating headers from slot arrays.

Object slot array The *object slot array* begins after the end of the mark array, at the first word with the required alignment. Object slots contain the payload of each page. At any given time each object slot may be either *used* or *unused*: when used it contains an object payload; when unused, its first word contains a pointer to the next unused object in the same page, or `NULL` in the case of the last unused slot.

For each page unused slots make up an independent free-list where elements are always ordered by address.

In order to avoid mistaking free list pointers in unused objects for pointers in used objects during conservative pointer finding, free list pointers are stored in *concealed* form by default[12].

Concealing consists in applying some function $c : \mathbb{A} \to \mathbb{A}^{\complement}$ to a free list pointer; it is important for c to be bijective, as concealing and then *unconcealing* (i.e. applying c^{-1} to) a pointer must preserve information.

c is trivially implemented as a C macro computing the successor function in `unsigned` (wrap-around) arithmetic: since its domain consists of word-aligned pointers, the elements of its image are guaranteed to be misaligned, hence they cannot be mistaken for pointers. The cost of applying either c or c^{-1} is one assembly instruction requiring no memory accesses[13].

Depending on the kind, some unused space may be present between the end of the mark array and the beginning of the slot array, and at the end of the page; in either case these two padding spaces are strictly smaller than the object effective size.

The global page table The global *page table* serves to recognize which part of the address space is being used for the garbage-collected heap; such information is important for avoiding dereferencing false pointers when doing conservative pointer finding.
Moreover, the collector needs to be able to recognize whether a heap pointer refers a kinded object in a page slot array or a large object — no particular provision is needed for kindless small objects, but we defer the justification of this fact to §6.3.5. Since we support interior pointers for large objects, it must also be possible to efficiently map an arbitrary (word-aligned) interior pointer to an initial pointer.

We call *candidate pointer* a word which is suspected to be a (possibly interior) object pointer at marking time, and *candidate page* the address of the hypothetical

[12]Free list pointer concealing can be disabled at configuration time.

[13]Assuming instructions such as either `inc`/`dec` or `add`/`sub` with a small *immediate* parameter; again, all modern machines satisfy this condition.

page which would contain the object referred by a candidate pointer. Of course candidate pages have alignment log_2`epsilongc_PAGE_SIZE_IN_BYTES`.

At an abstract level, the table implements a function f mapping a non-`NULL` candidate page p to an element s of the disjoint sum

$$Sort \triangleq \{kinded\} + \{nonheap\} + LargeObjects$$

If $f : p \mapsto kinded$ then the candidate page p is actually a page; if instead $f : p \mapsto nonheap$ then p is a pointer referring some object out of the garbage-collected heap, or a false pointer. Otherwise $f : p \mapsto l$, where l is the address of the beginning of the large object containing the word pointed by p.

Given a value for p stored as a key, a simple encoding allows us to represent any element of $Sort$ in a single word: `NULL` represents $nonheap$, $s = p$ stands for $kinded$, and any other value of s is interpreted as a large object pointer.

The table is implemented as a simple resizable chained hash where the first element of each bucket is stored within the bucket pointer array itself[14], as first described in [97]; the hash function is modulo.

One essential optimization at mark time consists in *not* consulting the page at all, which would be comparatively expensive, for `NULL` or misaligned candidate pointers.

It is interesting to notice how all *updates* to the global page table occur at mutation time, when creating or destroying[15] pages and large objects; unfortunately such updates require critical sections which, short as they are, may nonetheless limit scalability. By contrast at collection time the table is only *read*, which allows us to completely avoid critical sections for table access during that stage.

6.3.2.1 Page creation

Creating a page involves allocating space from the C heap, filling the header fields, initializing the mark and object slot arrays and registering the page in global structures.

Because of the alignment requirements we currently allocate pages with `posix_memalign()`[16]; as this may involve a kernel call and/or synchronization in the C library, such operation tends to be both expensive and hard to parallelize.

[14] This optimization is the reason why we don't include `NULL` in the domain of f: we use the value `NULL` as a key in a hash table element out of the bucket to mean that the element is currently unused.

[15] See §6.3.6 for the reason why pages must be destroyed at *mutation* rather than collection time.

[16] An interesting alternative to explore would involve using `mmap()` to allocate a group of pages; for some (non-GNU) implementations of `posix_memalign()`, the `mmap()` solution might incur a significantly lower space overhead, at the cost of always involving the kernel in page creation. Using `mmap()` could in fact make deallocation more portable, as `free()`ing buffers allocated with `posix_memalign()` is only permitted on GNU systems, as far as we know ([48], "Allocating Aligned Memory Blocks", currently at subsection 3.2.2.7).

However the `mmap()` solution has some issues of its own: `mmap()` only guarantees `sysconf(_SC_PAGESIZE)` alignment, hence pages could only be reasonably `mmap`ped in large groups, with some

Filling the header involves little more than copying some fields from the kind data structure, which is directly referred by the source, and making the free-list head point to the payload beginning. Nothing of this is performance-critical.

The mark array has to be zeroed at creation, with a `memset()` call. This should be relatively efficient, just involving some evictions from L1 — however having the mark array in L1 at page creation time does not buy us anything, as mark arrays are only touched during collection. If out-of-page mark arrays are enabled then we should add a `malloc()` call to the cost.

Building the free list involves some memory traffic, as all objects need to be touched. Unless objects have effective size larger than a cache line the complete object slot array has to be brought into cache. Even if this phase by itself is expensive, it may work like a sort of prefetching: if the page is used soon, all of it will already be loaded at least in the L2 cache.
We define *backward free list building*[17] the strategy of building the free list starting from the *last* slot which will be used for allocation. This solution has locality advantages in case of large page size, under the assumption that a just-created page will be used soon for allocating: if the page size is larger than the L1 data cache, building the free list backwards makes it very likely that the memory touched first while allocating will be already in L1; the rest of the page will be still in L2. It is possible to choose between forward and backward free list building at configuration time.

The final step is registering the page in the page table, which requires a critical section on the global mutex, plus a `malloc()` call within the critical section in case of hash collision.
All of this makes page creation a relatively expensive and non-scalable operation.

6.3.2.2 Page sweeping

Sweeping can be performed on an individual page without need for synchronization or kernel calls. It simply involves scanning the mark array and, for each i-th element, either clearing the corresponding element if $array[i]$ is one, or making the i-th object slot *unused* by re-adding it to the free-list if $array[i]$ is zero. Since the mark array is examined in order (either forward or backward, as per the free-list building

space overhead at the beginning and the end. Making `epsilongc_PAGE_SIZE_IN_BYTES` equal to `sysconf(_SC_PAGESIZE)` would solve the space overhead problem, but at the price of forcing pages to be unacceptably small. `unmmap`ping space from the middle of a `mmap`ped buffer is supported, but deallocation of single pages would still be a problem unless `epsilongc_PAGE_SIZE_IN_BYTES` were chosen to be a multiple of `sysconf(_SC_PAGESIZE)`. Re-`mmap`ping a previously `unmmap`ped part of a buffer is *typically* supported, even if such behavior is not mandated by POSIX. In addition we would need some data structure to keep track of which pages in a large buffer are `mmap`ped at any given time.
Anyway, despite all the complexity, such an idea seems worthy of some exploration.

[17] The actual direction of free list building, from higher addresses down to lower ones or from lower addresses up to higher ones, has no effect on performance as long as it is *the opposite* of the allocation direction: note in particular how automatic hardware prefetching works in either direction on modern processors ([23], section 3.3.2, “Single Threaded Sequential Access”).

direction), free list elements are kept ordered by address in the list. All the words of dead objects other than the first one are overwritten[18], to prevent future false pointers referring the slot to keep alive the objects which were referred by the now dead slot.

Memory access patterns in sweeping are similar to the ones in mark array initialization and free-list construction; in particular a just-swept page will likely remain cached at least in L2 — and the next lines to be used will be in L1, if backward free list building is enabled.

6.3.2.3 Page refurbishing

It is possible to re-use an empty page of some kind for objects of another kind: such operation is called *refurbishing*, and involves reconstructing the header, mark array and free list.

Refurbishing has essentially the same overhead as sweeping, and the cache effects of the two operations are also comparable: allocations from a just-refurbished page on the same thread which performed the refurbishing is efficient as all the page cache lines will still be in L1 and L2.

6.3.2.4 Page destruction

Destroying a page involves its deallocation and removal from the global page table: such operations are expensive and non-scalable, involving synchronization and possibly kernel calls.

6.3.3 Sources

From the implementation point of view a source is quite a trivial structure, serving as repository of pages. Each source simply contains two lists of pages, the *full pages list* and the *non-full pages list*, plus a mutex for synchronizing access to such lists.

6.3.4 Pumps

Pumps are performance-critical structures whose purpose at the implementation level consists in caching frequently accessed data about the objects to allocate. Such criticality is evident from the API in Figure 6.1, showing how existing pump data structures are *initialized* rather than dynamically allocated, in an effort to save a pointer indirection at runtime: pumps are conceived to be declared in programs as `__thread` variables of type `struct epsilongc_pump`, rather than as pointers.

At any given moment a pump may conceptually "contain" a page reserved to the allocating thread, or no page; of course at the implementation level such an inclusion

[18] Each word is overwritten with a configuration-dependent value impossible to mistake for a pointer: either the `0xdead` constant (which is easy to recognize for humans) if the collector is configured in debug mode, or otherwise simply `0` (which might lead to a slightly more efficient implementation on some architectures, possibly saving a *load immediate* instruction). Overwriting dead slots can also be completely disabled at configuration time.

is represented with a page pointer field. Its other relevant field is the current head of the page free list, again kept in the pump rather than in the contained page in order to avoid a pointer indirection at allocation time: in fact the free-list head field of the page is, counter-intuitively, *not* updated at each allocation. The free-list head field of the *pump* is set to `NULL` when the pump contains no page.

6.3.4.1 The allocation function

Despite the allocation being the only user-level operation on a pump, such a functionality is very performance-critical. Allocating from a given pump involves unconcealing the free-list field into a temporary variable, if non-`NULL` dereferencing it, setting the free-list head to the just loaded value and finally returning the temporary. This shorter and far more common execution path is carefully optimized and costs about *ten assembly instructions*, with no taken[19] jumps; the other execution path is taken in case of *page change* time, when a page is filled and another one must be acquired from the relevant pool, or at the first allocation for a pump with no page: it involves synchronization with the pool mutex and access to its lists. If no non-full pages are available, a page is taken from a *global empty pages list* (at the cost of one further synchronization) and refurbished if needed. If no empty pages are available, an heuristic is employed to decide whether to create a new page, or to trigger a collection. Page change is also the taken as the occasion for destroying empty pages, if an heuristic says that there are more than enough: the rationale here is to avoid destroying pages too frequently, since they might be needed again and both creation and destruction are expensive.

Repeatedly allocating from a page which was recently swept by the same thread and which contains many unused slots should be cache-friendly: sweeping works like a prefetch phase to load the page payload into the L1 or L2 cache, and even without on-demand sweep the hardware automatic prefetch may be activated when there is much free space on the page, as consecutive addresses are generated. Using pumps automatically guarantees that a page is only used for allocation by one CPU at a time, which avoids cache ping-pong.

6.3.5 Kindless and large objects

The data structures and primitives shown above provide no hints about the implementation of kindless objects, yet the idea is quite simple. A set of *implicit kinds*, *sources* and per-thread *pumps*[20], of user-definable sizes, are automatically defined: in this sense most kindless objects are just kinded objects "in disguise", only slightly less efficient because of the need for mapping an object size to a pump at runtime, and because of the possibility of internal fragmentation: not all possible sizes will be realistically provided, so the allocation of an object of a given size might be

[19] It is worth to provide GCC with an optimization hint via `__builtin_expect()`.

[20] Implicit pumps are created at thread registration and destroyed at thread un-registration time.

satisfied by using a larger buffer. For each size two kinds are provided, one with a fully conservative tracer, and another one with a leaf tracer (called "atomic" in the jargon of Boehm's collector).

It is easy to see how the solution above is not completely general, as it cannot satisfy allocation requests for objects larger than a page or even just larger than the maximum implicit kind size which has been fixed by the user. A different mechanism is provided for *large objects*, which are simply allocated one by one with `malloc()` and destroyed with `free()`. Their implementation is simple-minded and quite inefficient in both space and time, which given the functional hypothesis should hopefully not be serious. Of course the user-level API completely hides the difference between implicitly-kinded and large objects.

6.3.6 Garbage collection

A collection is initiated by one mutator, which stops all the other mutators with a signal. This choice has the advantage of allowing a simple user API, but significantly complicates the collector implementation: any function not reentrant with respect to signals, notably including `malloc()` and `free()`, can not be used at collection time: this is the reason why empty pages have to be destroyed at *mutation* rather than collection time.

The collection phase may internally proceed in two different orders according to a configuration option: if *on-demand sweeping* is enabled, as per the default, the three sub-phases are *non-deferred sweeping, root marking and marking*, otherwise they are *root marking, marking and sweeping*. In any case it is central to maintain the invariant according to which a complete heap marking is followed by a complete sweeping, before the next marking can begin.

On-demand sweeping consists in sweeping a page during mutation at page change time, *just before allocation from it begins*: such a choice is more cache-friendly than the traditional *stop-the-world sweep*, but it may leave some pages still to be swept when a collection begins: the non-deferred sweeping sub-phase, typically very short, serves to sweep such remaining pages. Non-deferred sweeping and stop-the-world sweeping share the exact same implementation.

After collection all mutators are restarted with a second signal.

Root marking Root marking is very simple, and currently *sequential*. Just like Boehm's collector in most of its configurations, it uses `setjmp()` for finding register roots in a portable way.

Marking Given the atomicity of mark array stores parallel marking can easily proceed in parallel without synchronization, if we accept the possibility of some (statistically unlikely) duplicate work; our implementation is quite canonical and closely follows Boehm's one [14], with load balancing in the style of Taura and Yonezawa [28]. It should be noted that the BiBOP organization does not affect marking in any significant way.

Sweeping Parallel sweeping is even simpler, with pages dictating the natural granularity for the operation of each thread: pages are simply taken from a list, swept and put back into another list.

6.3.7 Synchronization

One interesting and possibly original detail involves our locking style: in order to prevent a collection from starting during a critical section at mutation time, a global *read-write lock* is locked for reading at mutation, before acquiring the relevant mutex: the collection triggering function, before sending the signal, locks the same read-write lock *for writing*.

6.3.8 Data density

The system internally measures object size and alignment in *machine words*, and one word is the minimum size of a kinded object which can be represented without padding, in absence of alignment constraints specified by the user; with an alignment greater than one word, it becomes necessary in some cases to add some padding space right after the object payload; we call the *effective size* of an object the sum of its size and its alignment padding.

Given a kind k of objects with alignment a_k and size s_k, we define the effective size e_k needed to store each object, and the corresponding *data density* d_k, the number of objects representable per word, as:

$$e_k \triangleq a_k \cdot \left\lceil \frac{s_k}{a_k} \right\rceil \qquad\qquad d_k \triangleq \frac{1}{e_k}$$

The definitions above intentionally disregard all the sources of memory overhead out of object slot arrays, including mark arrays and all garbage collector data structures, the rationale being that density is not meant as a measure of memory occupation, but rather as an index of *the number of objects fitting in a cache line*: as mark arrays and other collector data structures are mostly accessed at different times from the objects *per se* and reside in different cache lines, optimizing data density maximizes the amount of useful information stored in the physically limited cache space at mutation time.

Data density may be reasonably defined in the same way independently from the garbage collecting strategy, and indeed it is of some interest to compare the values of d_k in different memory management systems for two kinds which are widely employed in functional programs, the *cons* (two words) and the non-empty *node* of an Red-Black binary tree of one given color[21] (three words: *left*, *datum* and *right*). Neither kind has alignment requirements, hence $a_{cons} = a_{node} = 1$.

Several systems such as the *GNU libc* `malloc()` facility [48], all the other allocators derived from Doug Lea's `malloc()` and — even more interestingly — Boehm's

[21] The example trivially generalizes to AVL trees, the idea being simply that the balance-related information can usefully be represented as *meta*-data rather than data.

collector [15], allocate all buffers at double-word-aligned addresses and may also add some internal status information *to each buffer*; metadata, when needed, must be represented as part of *each* object, adding to s_k. Instead many other systems, including just for example OCaml, do not force any alignment but always add one header word per object[22], sufficient to include a short tag, which again we consider part of s_k.

If metadata are accessed at runtime, as it is the case with dynamically-typed languages, with Boehm's collector we have $d_{cons} = d_{node} = \frac{1}{4}$. When metadata are not needed Boehm has optimal density in the cons case with $d_{cons} = \frac{1}{2}$, but again $d_{node} = \frac{1}{4}$. In OCaml, with or without metadata, $d_{cons} = \frac{1}{3}$ and $d_{node} = \frac{1}{4}$.

Independently of the need for metadata at runtime our model allows us to reach optimal density for both kinds, with $d_{cons} = \frac{1}{2}$ and $d_{node} = \frac{1}{3}$.

The data density of a particular representation seems likely to play a role in the *overall* efficiency of the system, even ignoring the cost of allocation and collection and considering only object accesses; anyway further empirical evidence will be needed to confirm this supposition for real world programs.

6.3.9 Closures

Functional programs written in certain styles[23] or CPS-transformed (§5.4.4.6) create a considerable number of short-lived closures at runtime. At a first look such a scenario does not seem to respect the functional hypothesis, as in principle closures can have many different shapes, depending on the number of non-locals captured in the environment, and on the fact that each non-local can be a pointer or a non-pointer.
Even if allocating all closures as kindless objects would work, the overhead of such a simple-minded solution is in fact easy to avoid.

First of all it should be observed that the great majority of functions need either zero or one variable in their non-local environment; it may be worth to add specific kinds for such common cases, and possibly also for the most performance-critical functions with larger non-local environments, when it is possible to recognize them with compile-time heuristics or after profiling.
The number of needed kinds can be reduced by establishing a convention for ordering non-locals in their environment arrays, according to whether they are pointers or not: either first all pointers then all non-pointers, or vice-versa.

The idea of *normalizing the representation* is a sort of pattern in the BiBOP scheme, generalizable to many other cases when using statically typed languages or ε personalities: there is no reason why two cases of different concrete types, possibly completely unconnected at a semantic label but with the same effective

[22] Some systems add even more than one header word per object. Sun's JDK, MMTk [11] and Microsoft's CLR, for example, use two words.

[23] And in particular when using simple compilers or interpreters: higher-order code can be simplified with flow analysis.

size and number of potential pointer fields, cannot be represented in such a way to share the same kind.

6.3.10 Lazy and object-oriented personalities

Lazy languages require a slightly more sophisticated data representation than call-by-value languages, as in a realistic implementation it must be possible to destructively update a still-unevaluated thunk, and replace it with the result at the end of its computation.
Unsurprisingly, `epsilongc` does not provide any support for changing the kind of an existing object while maintaining its identity; that could be possible *at collection time* in a moving scheme, but not with mark-sweep[24].

Any standard solution already employed by the collectors for lazy languages such as Haskell can be adopted: unfortunately some of the cleanness of the BiBOP model is lost in this case, as dataneeds to be tagged with at least a boolean (two in a concurrent environment: objects may be *thunks*, *in flux* or *ready*) recording the evaluation state of an object; any unused bit sequence in the payload or even the mark array entry of the object can do the job.

Accessing possibly still-to-be-evaluated objects will often require a conditional at runtime, just like in conventional implementations of lazy languages; after an object is known to be ready, BiBOP metadata can be accessed just as for eager languages.

Such a solution also necessarily requires some form of synchronization if the mutator threads are more than one: of course it is always possible to add a synchronization word in the payload, if needed.
From this point of view the situation is not different for "managed" languages such as Java, where *each* object contains a header word reserved for that purpose; yet we believe that not forcing such an expensive representation for *all* objects is preferable in the general case; the user can always implement some additional logic where needed, out of the memory management system *per se*.

For most runtimes there is no reason for keeping *one mutex per object*, and even lazy languages such as Haskell normally employ *strictness analysis* to statically recognize many cases in which laziness is not needed, and more efficient traditional representations can be safely used.

The work about *Prolific Types* [77] is relevant for object-oriented languages.

6.4 Status

`epsilongc`'s implementation totals around 5000 lines of heavily commented C code,

[24] In a moving BiBOP collector otherwise similar to `epsilongc` it might be reasonable to split each kind into a *evaluated* kind, plus a *thunk-or-evaluated* one: all the alive evaluated objects of a thunk-or-evaluated kind would be *re-kinded* at collection time. This idea does not look particularly hard to implement, but keeping the collector both efficient and language-agnostic might be challenging. Moving-time hooks definable by the user would solve the problem, at some cost; the overhead could be reduced by allowing to re-compile the hooks *as part of* the collector, to be called as inline functions, like described for example in [95].

quite easy to understand for being such a low-level concurrent piece of code not sparing C macros, `#ifdef`s, GCC function attributes and intrinsics in order to be support *Autoconf* options and be as general and efficient as possible.

In preliminary micro-benchmarks (`http://ageinghacker.net/publications/gc-draft.pdf`) `epsilongc` appears to perform better than Boehm's collector; anyway no realistic parallel workload has been measured, and we feel that our conclusions can only be tentative in this respect.

Our collector is *not* currently used by ε: since the current implementation of ε still relies on Guile for the s-expression frontend (§5.4.5) it also shares its memory management system, which has been a custom sequential mark-sweep garbage collector up to Guile 1.8.x, then replaced with Boehm's collector in the new series Guile 2.0.x. We expect to integrate `epsilongc` into ε as soon as we drop the dependency on Guile, or when we write a compiler — which can be done even while keeping the interactive system linked with Guile without re-implementing the frontend, at the cost of not having access to the frontend from compiled code.

Support for mutexes and other imperative synchronization features is trivial to add to ε's implementation with C primitives.

Exploiting the BiBOP organization from ε_1 does not appear particularly problematic. It will be interesting to test the benefits of the BiBOP strategy in code strongly based on sum-of-products (§5.4.4.3), *such as in complex transforms*; the necessary changes in the representation of sum tags do not seem very involved.

`epsilongc` will work as it is with ε, but in the longer term we plan to turn the current mark-sweep collector into the old generation of a generational system, where the younger generation is copying; this will be particularly relevant for the allocation patterns of CPS code, which tends to produce short-lived objects at a high rate. Implementing a collector which can be interrupted at any time by signals has been a fun and instructive challenge, but in the future we plan to seize the opportunity of coping with a moving collector in the young generation to introduce safe points. `epsilongc` also needs a couple of new functionalities, the most urgent of which are support for *finalization* and *weak pointers*.

The `epsilongc` sources have been committed to the main ε repository (see `https://savannah.gnu.org/bzr/?group=epsilon`)[25], as an independent subdirectory with its own build system.

Like the rest of the system it is free software, released under the GNU GPL version 3 or later [31].

[25] 2015 note: the GNU epsilon repository no longer uses bzr, and is now managed with git: see §5.4.5.

6.5 Summary

We implemented `epsilongc`, a parallel mark-sweep conservative-pointer-finding garbage collector for multicore machines. Conceived for ε, it is general enough to be used by other systems as well.

In order to exploit the memory hierarchies of modern machines, we pack data in a dense way without prefixing every object with a header, segregating objects by memory representation in a BiBOP organization.

This solution is most appropriate for functional personalities, in which most objects belong to one of a small set of kinds.

Conclusion

We formally specified and implemented a practical *extensible programming language* based on a very small first-order imperative core, plus powerful syntactic abstraction features: Lisp-style *macros* map user s-expression syntax into expression data structures; user-specified *transforms* permit arbitrary code-to-code transformation, with the intent of supporting extended syntactic features which are gradually "transformed away" into core forms. This open-ended approach enables research and experimentation.

As examples of the power of our extension mechanisms, we used transforms to implement *higher-order lexically-scoped anonymous procedures* and *first-class continuations*, on top of a core language only supporting named global procedures.

The language is very expressive and permits *reflection* and *self-modification*; it is possible to update the global state of the system by global modifications, possibly up until a state where the program is "static", convenient for analysis and compilable with traditional techniques.

We formally developed an analysis for static programs, and proved a *soundness* property about it with respect to the dynamic semantics. We argue that such formal reasoning is only possible thanks to the size and simplicity of the core language.

The state of the system can be saved and restored with *unexec* and *exec* facilities based on marshalling.

The language supports asynchronous threads and is suitable for modern multi-core machines. We implemented a parallel garbage collector, not yet integrated in the system, to limit garbage collection bottlenecks.

The implementation is not mature yet, but can be played with. The bulk of the system is written in itself, using C for the runtime, and Guile as a temporary dependency for bootstrapping.

An official part of the GNU project, epsilon is free software, released under the GNU GPL version 3 or later [31]. Its home page is `http://www.gnu.org/software/epsilon`.

The source code is managed on a public bzr server[26], and a public mailing list is available for discussion: see `https://savannah.gnu.org/projects/epsilon` for more information.

[26] 2015 note: the repository switched from bzr to git in late 2013: see §5.4.5.

Bibliography

[1] Harold Abelson, Norman Adams, David Bartley, Gary Brooks, William Clinger, Dan Friedman, Robert Halstead, Chris Hanson, Chris Haynes, Eugene Kohlbecker, Don Oxley, Kent Pitman, Jonathan Rees, Bill Rozas, Gerald Jay Sussman, and Mitchell Wand. The Revised Revised Report on Scheme or An UnCommon Lisp. AI Memo 848, MIT, 1985.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996. 6

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007. 78

[4] ANSI. *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. American National Standards Institute, 1996. Available in a hyperlinked version as *The Common Lisp HyperSpec* at `http://www.lispworks.com/documentation/HyperSpec/Front`. 8, 19, 73, 81, 95, 101

[5] Andrew W. Appel. *Compiling with Continuations*. Cambridge Univ. Press, 1991. 10, 53, 118, 122

[6] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. 11

[7] Alan Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Program Manipulation*, pages 4–12, 1999. 116

[8] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. *SIGPLAN Not.*, 34:129–140, September 1998. 19

[9] Gérard Berry and Laurent Cosserat. The *Esterel* synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer-Verlag, 1984. 3

[10] Richard J. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, second edition, 1998. 5

[11] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS'04, Performance Evaluation Review (PER)*, pages 25–36, New York, NY, USA, June 2004. ACM SIGMETRICS/IFIP. 140

[12] Hans-Juergen Boehm. Fast multiprocessor memory allocation and garbage collection. Technical Report HPL-2000-165, Hewlett Packard Laboratories, December 21 2000. Available at `http://www.hpl.hp.com/techreports/2000/HPL-2000-165.html`. 126, 132

[13] Hans-Juergen Boehm. "Re: Unregistering the main thread". Thread on Boehm's GC public mailing list, September 2008. `http://www.hpl.hp.com/hosted/linux/mail-archives/gc/2008-September/002334.html`. 126

[14] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *PLDI*, pages 157–164, 1991. 138

[15] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, September 1988. 126, 140

[16] Maximilian C. Bolingbroke and Simon L. Peyton Jones. Types are calling conventions. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 1–12, New York, NY, USA, 2009. ACM. 19

[17] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *IEEE Transactions on Software Engineering*, 22(4):256–266, 1996. 3

[18] Luca Cardelli. Basic polymorphic typechecking. *The Science of Programming*, 8(2):147–172, 1987. 3

[19] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications with Objective Caml.* O'Reilly, 2000. Full text available at `http://caml.inria.fr/pub/docs/oreilly-book`. 4

[20] William Clinger and Jonathan Rees. Revised[4] Report on the Algorithmic Language Scheme. *ACM SIGPLAN Lisp Pointers*, 4(3):1–55, July/September 1991. Available at `ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/standards/r4rs.ps.gz`. 57

[21] Ludovic Courtès, Andy Wingo, Neil Jerram, Jim Blandy, et al. *Guile 2.0.5 Reference Manual*, January 2012. Full text available at `http://www.gnu.org/software/guile/manual`. Also available on paper from Network Theory, edited by Brian Gough: `http://www.network-theory.co.uk/guile`. 13, 49, 86, 102

[22] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.7528`. 57, 63

[23] Ulrich Drepper. What every programmer should know about memory. Technical report, RedHat, November 2007. 126, 135

[24] R. Kent Dybvig. *Three Implementations Models for Scheme*. PhD thesis, University of North Carolina, April 1987. Full text available at `http://www.cs.indiana.edu/~dyb/pubs/3imp.pdf`. 119

[25] R. Kent Dybvig. Syntactic abstraction: The `syntax-case` expander. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly and Associates, 2007. Available at `www.cs.indiana.edu/~dyb/pubs/bc-syntax-case.pdf`. Included in [62]. 13, 152

[26] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible, and efficient storage management for dynamically-typed languages. Technical Report TR 400, Indiana University, Computer Science Department, March 1994. Available at `http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR400`. 127

[27] R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. Technical report, Indiana University, 2005. Available at `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.68.9352`. 123

[28] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *High Performance Computing and Networking (SC'97)*, 1997. 138

[29] Anton Ertl et al. Forth 200x web page, 2012. An underway effort to update the 1994 Forth standard. `http://www.forth200x.org/forth200x.html`. 13, 149

[30] ANS Forth Technical Committee. *ANS Forth Standard – document X3.215-1994*. American National Standards Institute, 1994. Available at `www.openfirmware.info/data/docs/dpans94.pdf`. A new process is now underway to update the 1994 Standard [29]. 13

[31] Free Software Foundation. GNU General Public License. Web page, 2007. 123, 142, 145

[32] Richard P. Gabriel and Kent M. Pitman. Endpaper: Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, 1(1):81–101, June 1988. Available at `http://www.nhplace.com/kent/Papers/Technical-Issues.html`. 95, 121

[33] Martin Gasbichler and Michael Sperber. Final shift for `call/cc`: direct implementation of shift and reset. *ACM SIGPLAN Notices*, 37(9):271–282, September 2002. Available at `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.3425`. 123

[34] GreenArrays. GreenArrays web page, 2010-2012. Headed by the father of Forth Chuck Moore [60], GreenArrays sells parallel chips for embedded applications implementing a custom Forth dialect *in hardware*. Moore designs Forth

processors, from transistor layout up, using CAD systems written by himself in Forth. `http://www.greenarraychips.com`, `http://www.colorforth.com/vlsi.html`. 13

[35] David Gudeman. Representing type information in dynamically-typed languages. Technical Report TR93-27, University of Arizona, Department of Computer Science, Tucson, AZ, 1993. Available at `ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/typeinfo.ps.gz`. 50, 102

[36] C. A. R. Hoare. Hints on programming language design. Technical Report AIM 224, Stanford AI Lab., December 1973. From §3.3:

> *(2)* `precompile`. This is a directive which can be given to the compiler after submitting any initial segment of a large program. It causes the compiler to make a complete dump of its workspace including dictionary and object code, [...]
>
> *(3)* `dump`. This is an instruction which can be called by the user program during execution, and causes a complete binary dump of its code and workspace into a named user file. The dump can be restored and restarted at the instruction following the dump by an instruction to the operating system.

This discusses a feature not unlike our *unexec* facility, brought forward in a very different context with the purpose of optimizing single-pass compilers. 49

[37] ISO. The ANSI C Standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999. 8, 19, 131

[38] ISO. Standard for Programming Language C++. Technical Report ISO/IEC JTC1/SC22/WG21, ISO, 2011. Previously known as "C++0x". A recent draft close to the final version (N3337, January 2012) is available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`. 7

[39] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International Series in Computer Science. Prentice Hall, 1993. Full text available at `http://www.itu.dk/people/sestoft/pebook/`. 85

[40] S. Peyton Jones, editor. *Haskell 98 Language and Libraries, the Revised Report.* Cambridge Univ. Press, April 2003. 5

[41] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.2271`. 57, 73, 87

[42] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In Richard P. Gabriel, editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 151–181, Cambridge, MA, August 1986. ACM Press. 13

[43] Philip J. Koopman, Jr. *Stack Computers.* Ellis Horwood Limited, 1989. Full text available at `http://www.ece.cmu.edu/~koopman/stack_computers/index.html`. 13

[44] David A. Kranz. *ORBIT: An Optimizing Compiler For Scheme.* PhD thesis, Yale University, New Haven, Connecticut, February 1988. 10, 122

[45] E. Ulrich Kriegel. A conservative garbage collector for an eulisp to ASM/C compiler. OOPSLA 1993 Workshop on Memory Management and Garbage Collection, September 1993. 127, 131

[46] Xavier Leroy. Functional programming languages – Part III: program transformations, 2007. A set of slides from a functional programming class, covering several program transforms in a very clear and accessible style. `http://gallium.inria.fr/~xleroy/mpri/progfunc/transformations.2up.pdf`. 10, 11, 122

[47] Bil Lewis, Dan LaLiberte, Richard Stallman, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual, for Emacs Version 23.3.* Free Software Foundation, Inc., Boston, Massachusetts, 3.0 edition, 2011. Available at `http://www.gnu.org/software/emacs/manual`. 8, 18, 48, 95

[48] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. *The GNU C Library Reference Manual.* GNU Press, 2006. 134, 139

[49] Dave MacQueen. `heap2exec`, 2007. `http://www.smlnj.org/doc/heap2exec/index.html`. 49

[50] Michel Mauny. *Functional programming using Caml Light (version 0.7).* INRIA, 1995. Full text available at `http://www.mauny.net/data/papers/mauny-1995b.pdf`. x

[51] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960. The famous paper introducing Lisp. McCarthy's 1995 revision suggests looking at [88] for `eval`, including the version which was actually implemented. Available at `http://www-formal.stanford.edu/jmc/recursive.pdf`. 46, 73, 76, 78, 81, 95, 127, 154

[52] W.M. McKeeman, J.J. Horning, and D.B. Wortman. *A compiler generator.* Prentice-Hall series in automatic computation. Prentice-Hall, 1970. 85

[53] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I. *Information and Computation*, 100(1):1–40, 1992. 4

[54] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part II. *Information and Computation*, 100(1):41–77, 1992. 4

[55] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. The famous paper introducing the $\mathscr{W}$ algorithm. 9, 69

[56] Robin Milner. *A calculus of communicating systems*, volume 92 of *Lecture notes in computer science*. Springer-Verlag, 1980. 4

[57] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, MA, USA, 1991. This is a commentary of [58], and Tofte explicitly wrote on his home page (`http://www.itu.dk/people/tofte/publ/1990sml/1990sml.html`) that it does not apply to [59]. Available at `http://www.itu.dk/people/tofte/publ/1990sml/frontcommentary.pdf` and `http://www.itu.dk/~tofte/publ/1990sml/1991commentaryBody.pdf`. 57

[58] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, August 1990. Available at `http://www.itu.dk/people/tofte/publ/1990sml/front1990sml.pdf` and `http://www.itu.dk/people/tofte/publ/1990sml/1990sml.pdf`. 57, 152

[59] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997. 57, 152

[60] Charles E. Moore. *colorforth.com*, 2012. Chuck Moore's home page. `http://colorforth.com`. 13, 149

[61] J. Moses. The function of FUNCTION in LISP, or, why the FUNARG problem should be called the environment problem. Report MAC-M-428 and A. I. MEMO 199, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, 1970. 4

[62] Andy Oram and Greg Wilson, editors. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly, 2007. This includes [25]. 149

[63] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002. 9

[64] Kent M. Pitman. Special forms in LISP. In *LISP Conference*, pages 179–187, 1980. 7, 12

[65] Jacques Pitrat. Implementation of a reflective system. *Future Gener. Comput. Syst.*, 12(2-3):235–242, 1996.

[66] Jacques Pitrat. *Artificial Beings (the conscience of a conscious machine)*. Wiley/ISTE, march 2009.

[67] Marc Pouzet. Lucid synchrone - version 2.0: Tutorial and reference manual, October 22 2001. 3

[68] Christian Queinnec. A library of high-level control operators. *Lisp Pointers, ACM SIGPLAN Special Interest Publ. on Lisp*, 6(4):11–26, October 1993. Available at `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.4790`. 123

[69] M. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959. This presents what is now known as the Rabin-Scott Construction. 123

[70] Jonathan A. Rees and William Clinger. Revised^3 Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, 21(12), December 1986. Available at `ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/standards/r3rs.ps.gz`. 57

[71] Didier Remy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998. 4

[72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM. 11, 87, 98

[73] Christophe Rhodes. SBCL: A sanely-bootstrappable Common Lisp. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems, First Workshop, S3 2008, Potsdam, Germany, May 15-16, 2008, Revised Selected Papers*, volume 5146 of *Lecture Notes in Computer Science*, pages 74–86. Springer, 2008. 46, 102

[74] Hovav Shacham, Eu jin Goh, Nagendra Modadugu, Ben Pfaff, and Dan Boneh. On the effectiveness of address-space randomization. In *In CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM Press, 2004. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.4638`. 51

[75] S. T. Shebs and R. R. Kessler. Automatic design and implementation of language data types. *SIGPLAN Not.*, 22(7):26–37, 1987. 102

[76] Olin Shivers. History of T, 2001. Available from `http://www.paulgraham.com/thist.html`.

[77] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. *ACM SIGPLAN Notices*, 37(1):295–306, January 2002. 141

[78] John N. Shutt. *Fexprs as the basis of Lisp function application or $vau: the ultimate abstraction.* PhD thesis, Worcester Polytechnic Institute, September 2010. Available at `http://www.wpi.edu/Pubs/ETD/Available/etd-090110-124904/`. 7, 12

[79] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. Revised^6 Report on the Algorithmic Language Scheme, September 2007. Available at `http://www.r6rs.org`. 8, 10, 12, 19, 57, 73, 116

[80] The SRFI Editors. Scheme Requests For Implementation. SRFIs, archived discussions and process documents are available at `http://srfi.schemers.org/`, 1998-2010. 6

[81] Richard Stallman and the GNU Emacs contributors. *GNU Emacs Manual*. Free Software Foundation, Inc., Boston, Massachusetts, Sixteenth (updated for Emacs version 23.3) edition, 2011. Available at `http://www.gnu.org/software/emacs/manual`. 86

[82] Guy Lewis Steele, Jr. Data representations in PDP-10 MACLISP. Report A. I. MEMO 420, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, 1977. Available at `http://dspace.mit.edu/handle/1721.1/6278`. 95, 127

[83] Guy Lewis Steele, Jr. Rabbit: A compiler for scheme. AI Technical Report 474, MIT Artificial Intelligence Laboratory, May 1978. 10, 122

[84] Guy Lewis Steele, Jr. *Common Lisp: the Language*. Digital Press, Bedford, Massacusetts, second edition, 1990. x

[85] Guy Lewis Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, October 1999. 5, 6, 14

[86] Guy Lewis Steele, Jr. A growable language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 505–505, New York, NY, USA, 2006. ACM. Slides are available at `http://labs.oracle.com/projects/plrg/Publications/OOPSLA-Growable-Language-2006public.pdf`. 6

[87] Guy Lewis Steele, Jr. and Gerald Jay Sussman. The revised report on SCHEME, a dialect of LISP. AI Memo 452, MIT, 1978. Available at `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-452.pdf`. 6, 7, 83

[88] Herbert Stoyan. *The influence of the designer on the design–J. McCarthy and LISP*, pages 409–426. Academic Press Professional, Inc., San Diego, CA, USA, 1991. A detailed analysis on early Lisp memos by McCarthy; McCarthy cited this article in his 1995 revision of [51], stating that the versions of `eval` shown by Stoyan included the one which was actually implemented. Available at `http://www8.informatik.uni-erlangen.de/html/lisp/mcc91.html`. 151

[89] Gerald Jay Sussman and Guy Lewis Steele, Jr. SCHEME: An interpreter for extended lambda calculus. Technical Report AI Memo No. 349, Massachusetts

Institute of Technology, Cambridge, UK, December 1975. Available at `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-349.pdf`. 6, 10, 73, 122

[90] Herb Sutter. The free lunch is over: a fundamental turn toward toward concurrency. *Dr. Dobb's Journal*, March 2005. 125

[91] Ken Thompson. Regular expression search algorithm. *Journal of the ACM*, 11(6):419–422, June 1968. 123

[92] Peter Van Roy. Programming paradigms for dummies: What every programmer should know. In *New Computational Paradigms for Computer Music*, pages 9–47. Delatour, 2009. 1

[93] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004. 4

[94] Stephen T. Weeks. MLton user guide, September 07 2000. 46

[95] Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, December 1998. 141

[96] Cheryl A. Wiecek. A model and prototype of VMS using the Mach 3.0 kernel. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 187–204, Seattle, WA, USA, April 1992. USENIX Association. 14

[97] F. A. Williams. Handling identifiers as internal symbols in language processors. *Communications of the ACM*, 2(6), June 1959. 134

[98] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994. 45, 49, 52, 96

[99] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993. 29

Titre en français GNU epsilon — un langage de programmation extensible

Résumé en français Le *réductionnisme* est une technique réaliste de conception et implantation de vrais langages de programmation, et conduit à des solutions plus faciles à étendre, expérimenter et analyser.

Nous spécifions formellement et implantons un langage de programmation extensible, basé sur un *langage-noyau minimaliste impératif du premier ordre*, équipé de *mécanismes d'abstraction* forts et avec des possibilités de *réflexion* et *auto-modification*. Le langage peut être étendu à des niveaux très hauts : en utilisant des *macros* à la Lisp et des *transformations de code à code* réécrivant les expressions étendues en expressions-noyau, nous définissons les clôtures et les continuations de première classe au dessus du noyau.

Les programmes qui ne s'auto-modifient pas peuvent être analysés formellement, grâce à la simplicité de la sémantique. Nous développons formellement un exemple d'*analyse statique* et nous prouvons une *propriété de soundness* par apport à la sémantique dynamique.

Nous développons un *ramasse-miettes parallèle* qui convient aux machines multi-cœurs, pour permettre l'exécution efficace de programmes parallèles.

Titre en anglais GNU epsilon — an extensible programming language

Résumé en anglais *Reductionism* is a viable strategy for designing and implementing practical programming languages, leading to solutions which are easier to extend, experiment with and formally analyze.

We formally specify and implement an extensible programming language, based on a *minimalistic first-order imperative core language* plus strong *abstraction mechanisms*, *reflection* and *self-modification* features. The language can be extended to very high levels: by using Lisp-style *macros* and code-to-code *transforms* which automatically rewrite high-level expressions into core forms, we define closures and first-class continuations on top of the core.

Non-self-modifying programs can be analyzed and formally reasoned upon, thanks to the language simple semantics. We formally develop a *static analysis* and prove a *soundness property* with respect to the dynamic semantics.

We develop a *parallel garbage collector* suitable to multi-core machines to permit efficient execution of parallel programs.

Discipline Informatique

Mots-clés programmation, langage, extensibilité, macro, transformation, reflection, *bootstrap*, interprétation, compilation, parallélisme, concurrence, ramasse-miettes

Intitulé et adresse du laboratoire

LIPN, UMR 7030 – CNRS, Institut Galilée, Université Paris 13

99, avenue J.-B. Clément

93430 Villetaneuse

France