Abstract

Principled Compilation and Scavenging

Stefan Monnier

With the growth of the Internet and of the use of computers in general, the safety and reliability of computer systems is much more important. As the most popular methods to address these needs, such as discipline, authentication, or monitoring, are proving insufficient, there grows a need for formal methods that enable automatic verification of the safety of computer programs.

While approaches such as model checking or Hoare logic hold the promise of proving not just safety but even correctness, their applicability is too restricted for general purpose applications. At the other end of the scale, type systems are very popular but only check a very limited set of properties and impose sometimes severe restrictions which can prevent some forms of optimizations and even whole categories of code.

In order to use type systems to guarantee the security of computer systems, applications need to ship with type annotations, which requires *type-preserving compilers*, i.e. compilers that preserve the code's type information through every stage of the compilation. As the program gets turned into lower level code, the restrictions imposed by the type system become more crippling, or alternatively, the type system needs to become more sophisticated.

My thesis defines and uses, within the context of type-preserving compilation, novel type systems for the purpose of proving more sophisticated properties of programs, thus trying to reduce the gap between program verification techniques and type systems. More specifically, I first present my experience porting the optimizer of the SML/NJ compiler to a type-preserving intermediate language. Then I extend this with a cross-module inliner and show a typed two-level language that allows me to formalize the inlining algorithm as a form of partial evaluation. In the second part, I develop techniques to type check a garbage collector and its interaction with the mutator. This leads me to develop a novel type system that allows me to reason about sophisticated invariants, much like Hoare logic would but without losing the modularity so crucial to the success of type systems.

Principled Compilation and Scavenging

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

By
Stefan Monnier

Dissertation Director: Zhong Shao

December 2003

Contents

Contents i

List of Figures vi

Acknowledgement viii

1 Introduction 1

1.1 The need for stronger type systems 2

1.2 Reasoning about the state 2

1.3 Optimizing a typed intermediate language 3

1.4 Formalizing a cross-module inlining algorithm 4

1.5 Type-safe garbage collection 4

1.6 Typed regions 5

2 The FLINT optimizer 6

2.1 The old middle end 7

2.2 The FLINT intermediate language 10

2.3 The FLINT optimizer 12

2.3.1 Known functions 13

2.3.2 General organization 13

2.3.3 Flattening 15

2.3.4 Contracting β-redexes 18

2.4 A-normal vs CPS 19

2.4.1 The difference 19
2.4.2 Advantages of using CPS 21
2.4.3 Advantages of A-normal 23
2.5 Impact of types 24
2.6 Related work 26
2.7 Conclusion 27

3 Inlining as staged computation 29

3.1 The big picture 32
3.2 The source calculus SRC 34
3.3 The target calculus TLC 37
3.4 Translation from SRC to TLC 41
3.4.1 A quick overview 42
3.4.2 Staging the core 43
3.4.3 Splitting 47
3.4.4 The main algorithm 52
3.5 Handling the top level 56
3.6 Extensions 59
3.6.1 Side effects 59
3.6.2 Recursion 61
3.6.3 Optimizations as staged computation 62
3.6.4 Staging refinement 63
3.6.5 Link-time optimizations 63
3.6.6 Implementation 63
3.7 Related work 64
3.8 Conclusions and future work 67

4 Principled scavenging 68

4.1 Motivation and approach 69

4.1.1 The problem 70
4.1.2 My solution 72
4.2 Source language λ_{CLOS} 75
4.3 Target language λ_{GC} 76
4.3.1 Regions 76
4.3.2 Functions and code 77
4.3.3 Intensional type analysis 78
4.4 Translating λ_{CLOS} to λ_{GC} 78
4.4.1 Closed CPS garbage collector 82
4.5 A closer look at λ_{GC} 82
4.5.1 Closure conversion and *copy*. 84
4.5.2 Functions and code 85
4.5.3 The type calculus 85
4.5.4 The term calculus 86
4.5.5 Formal properties of the language 87
4.6 Forwarding pointers 91
4.6.1 How to widen safely 94
4.7 Generational collection 97
4.8 Related work 100
4.9 Conclusion 102

5 Typed regions **103**
5.1 Overview 105
5.2 Background 106
5.2.1 Cyclic structures 106
5.2.2 Regions 108
5.2.3 Alias types 110
5.2.4 Calculus of inductive constructions 112
5.3 Typed regions 113

5.3.1 The core language . . . 114
5.3.2 Semantics . . . 116
5.3.3 Properties of the core language . . . 120
5.4 Examples . . . 121
5.4.1 Simple regions . . . 121
5.4.2 Stacks . . . 123
5.4.3 *Adoption* and *focus* . . . 124
5.5 The full language . . . 126
5.5.1 Semantics . . . 128
5.5.2 Properties of the full language . . . 132
5.6 Generational collection . . . 132
5.6.1 Simple type-preserving GC . . . 132
5.6.2 Generational GC . . . 133
5.6.3 Solution . . . 136
5.7 Related work . . . 138
5.8 Conclusion . . . 140

6 Future directions **141**
6.1 Implementation . . . 141
6.2 Advanced uses . . . 142
6.3 Surface syntax . . . 142

Bibliography **143**

A Soundness proofs **150**
A.1 Soundness of principled scavenging . . . 150
A.1.1 Soundness of λ_{GC} . . . 150
A.1.2 Soundness of $\lambda_{\mathsf{GCforw}}$ . . . 165
A.1.3 Soundness of λ_{GCgen} . . . 173
A.2 Soundness of typed regions . . . 184

A.2.1 Soundness of the core language 184
A.2.2 Soundness of the full language 194

List of Figures

2.1 The FLINT datatype . . . 10

2.2 Flattening polymorphic code. . . . 17

3.1 Structure of the compiler . . . 32

3.2 Syntax for the source calculus SRC . . . 34

3.3 Static semantics for SRC . . . 35

3.4 Syntax for the target calculus TLC . . . 37

3.5 Static semantics for TLC . . . 39

3.6 Primitive static reduction for TLC . . . 40

3.7 Single-step partial evaluation for TLC . . . 40

3.8 Primitive reduction relation for TLC . . . 40

3.9 Single-step call-by-value standard evaluation for TLC . . . 40

3.10 Core type translations $|\cdot|$ and coercions $\downarrow$ and $\uparrow$. . . . 45

3.11 Core code translation. . . . 46

3.12 The λ-split algorithm. . . . 47

3.13 Type and environment translation. . . . 52

3.14 Binding-time coercions. . . . 53

3.15 Module code translation. id_t^b is a shorthand for $\lambda^b x{:}t.x$. . . . 54

3.16 New rules using existential types. . . . 60

4.1 Stop-and-Copy from region F to region T. . . . 71

4.2 Syntax of λ_{GC} . . . 77

4.3 Translation of λ_{CLOS} terms. . . . 79

4.4 The garbage collector proper. . . . 81

4.5 The basic GC code after CPS and closure conversion. . . . 83

4.6 Operational semantics of λ_{GC}. . . . 84

4.7 Static semantics of λ_{GC}, part 1. . . . 88

4.8 Static semantics of λ_{GC}, part 2. . . . 89

4.9 Environment formation rules. . . . 90

4.10 Static semantics for $\lambda_{\mathsf{GCforw}}$. . . . 92

4.11 GC with forwarding pointers. . . . 95

4.12 Static semantics of λ_{GCgen}. . . . 98

4.13 Generational GC. . . . 100

5.1 Syntax of a region-based language. . . . 108

5.2 Syntax of an alias-types language. . . . 110

5.3 Syntax of the core language. . . . 114

5.4 Operational semantics of the core language. . . . 116

5.5 Environment formation rules. . . . 117

5.6 Static semantics of the core language. . . . 119

5.7 Syntax of the full language 126

5.8 Operational semantics of the full language. . . . 127

5.9 Static semantics of the full language: auxiliary judgments. . . . 129

5.10 Static semantics of the full language: well-formed terms. . . . 130

5.11 Generational GC 137

Acknowledgement

My work was funded in part by Yale University fellowship, NSF grants CCR-9633390, CCR-9901011, CCR-0081590, and CCR-0208618, and a DARPA OASIS grant F30602-99-1-0519.

I owe a lot of thanks to too many people to list them all here.

Let me specifically mention my officemate Chris League who somehow managed to survive my constant rambling about Emacs; Bratin Saha who is really responsible for the bulk of the soundness proofs for Chap. 4; Valery Trifonov for his endless supply of explanations about any kind of formal system; and of course my advisor Shao Zhong for his enthusiasm, his unexplained trust in my abilities, and his constant support.

Of course, this work would never have seen the light of day without all the feedback and comments I got on all my submitted papers.

Thanks also to all my friends, among whom Dominik stands out as the most closely related to this work. Finally, I would like to thank Fernanda for not beating me over the head, even in the last few days before my defense, and without whom this effort seemed to lack meaning.

Chapter 1

Introduction

The goal of my thesis is to show that type systems can be used to guarantee more sophisticated properties than basic type safety. More specifically, I present in the first part a typed language in which I can prove some behavioral properties of a cross-module inliner and in the second part, I describe how to use type systems to show that a generational copying garbage collector (GC) is correct in the sense that it will only deallocate memory that the mutator will not access in the future.

Type systems are often viewed as ways to do sanity checking of source code, as a guide to help design and structure code, or as instructions to determine the implementation strategy used by the compiler. Type annotations can also be considered as specifications which are automatically checked by the compiler.

The limitations imposed by a type system manifest themselves either in that the type system rejects code that should be considered valid (i.e. it is too restrictive), or in that it accepts code which should be considered incorrect (i.e. it is too permissive). Very often those two situations, although apparently diametrically opposed, are just the two sides of the same coin: by being too permissive in one case, the system cannot verify the invariant necessary to accept the other.

Traditional type systems such as used in SML or Java impose some non-trivial restrictions, but are very popular nonetheless because a large class of programs does not suffer terribly from them whereas they provide a welcome safety cushion in return for a little bit of extra annotation. Yet there is a growing need for more flexible type systems that can capture more sophisticated invariants and thus impose fewer restrictions.

1.1 The need for stronger type systems

This need is coming mostly from growing security problems with the use of the Internet, but also from the fact that the field of computer systems is slowly maturing to the point where people consider bugs not just as something you have to learn to live with but as a problem that needs to be addressed.

With regards to the more pressing security problem, a popular solution is indeed to rely on type systems to ensure the basic safety of the system. The best known examples of this are Java and .NET. But in order to be able to type-check code before running it, you need the program to come in a form that lends itself to type-checking. Both of those systems solve this by keeping the "compiled" code at a fairly high level, leaving the rest of the work to a just-in-time compiler. The reason for it is that when trying to type-check lower-level code, the invariants necessary to show its safety become more complex. Or rather, as details of the implementation are made explicit in the lower-level code, details of the proof of safety need to be made explicit in the types.

The field of *certifying compilation* aims at solving those difficulties up to the point where compilers can generate an actual proof of safety along with the fully optimized raw binary code [Nec97, MWCG98, App01, HST$^+$02]. The proof is typically encoded in some Hoare logic, so that it is fully generic and independent from any type system the compiler might have chosen to use.

1.2 Reasoning about the state

One of the more challenging difficulties for type systems is to express properties of the state. Traditional type systems only reason about properties of values. The problem with reasoning about the state is that when a memory cell is modified, it can influence invariants at any number of places, due to aliasing: there might be any number of pointers to this memory cell. Solutions to this problem traditionally rely on keeping track of all the aliases, which is not possible in the general case.

One of the contributions of this thesis is to show another way to tackle this problem which relies on preserving invariants that do not depend on aliasing. I claim that this is needed for certifying compilers because the source language in general does not impose any constraint on the aliasing patterns, so the compiler cannot use any of the approaches that relies on static aliasing information if it needs to reason about global state.

The rest of this chapter gives an overview of the thesis.

1.3 Optimizing a typed intermediate language

The FLINT project [Sha97b] uses the SML/NJ compiler [AM91] as a testbed for its work on certifying compilation. For many years now the front-end generates strongly typed intermediate code, but the type information was mostly thrown out just before going through the main code optimizer.

Obviously, if we want to preserve types throughout the compilation, we need to preserve them through optimization, since a non-optimizing compiler would be even academically of dubious interest. So I ported all those optimizations from the untyped intermediate language (called CPS) to the typed language (called FLINT). These optimizations are fairly simple. They include inlining, η-contraction, uncurrying, argument flattening, constant propagation, and constant folding, and cross-module inlining.

Chapter 2 reports on the problems we have encountered and how we solved them. CPS and FLINT also differ in a few more points than types, so I also report on my experience with those aspects of the respective languages.

1.4 Formalizing a cross-module inlining algorithm

The cross-module inliner I implemented is based on the work of Matthias Blume [BA97] but with the difference that his work was done in an untyped setting and with a few subtle differences in the behavior.

While arguing about what the proper behavior of the inliner should be and how to guarantee it, we found a need for a formalism in which we could express the algorithm such that we can try to reason about some of its properties.

Chapter 3 develops such a formalism in the form of a language in which the type of each expression is annotated with the phase (either compile-time or run-time) in which it is evaluated. I then present how to translate a simple SML-like source language into this intermediate language. Finally I formalize the cross-module inlining algorithm and show some of its properties.

1.5 Type-safe garbage collection

The second part of my thesis, starting with Chap. 4 presents techniques to write a garbage collector in a type-safe language.

Certifying compilers typically take as input programs written in a language using a traditional type system, which relies on automatic memory reclamation. So the generated code needs to come with enough annotations to show that the memory management is done correctly. The first step in this direction was made by Daniel Wang who shows in [WA01] how to implement a very simple copying garbage collector (GC) on top of a type-safe language with region annotations.

In this chapter I extend this work to address some of its limitations. First, in order to keep the type system simple, his work requires whole program analysis, forces the garbage collector to be tailored to the application, and requires every type used by the application to come with its own set of garbage collection routines. I address these issues using a more sophisticated type system that allows intensional type analysis.

In this chapter I also show two extensions to the basic GC: first, using a very simple extension of the type system, I show how to encode in the types the age-barrier invariant of a simplistic generational garbage collector. Second, I develop a first, very primitive and ad-hoc, attempt at dealing with issues of global state, which I use to implement forwarding pointers.

1.6 Typed regions

The above work is based on well understood type systems, with minor extensions. But all the GCs one can implement in such systems suffer from very serious limitations. The main limitation they all share is that they are unable to deal with cycles in the heap.

It turns out that garbage collecting a heap that includes cycles is almost impossible with such simple type systems. It requires more advanced reasoning about the state. In short, creating a cycle requires a memory assignment and we need to be able to reason about properties of this assignment in order to show that the GC will only collect garbage.

Chapter 5 presents a novel type system that allows such reasoning about the state, even in the absence of any aliasing information. I first explain why such reasoning is required and why existing type systems are unable to solve the problem in its generality. Then I show how this type system can be used to implement stack allocation as well as a generational GC. Contrary to the generational GC presented in the anterior chapter, this collector does not suffer from any fundamental restriction.

Chapter 2

The FLINT optimizer

Standard ML [MTHM97] is a modern functional programming language, with a flexible type system and a powerful module language. SML/NJ [App91] is a robust and mature implementation of the Standard ML language which is also used as a testbed for research in compiler technology.

Since version 109.24, the SML/NJ compiler includes a strongly typed intermediate language which is used to implement some type-directed optimizations, most importantly a flexible representation analysis [Sha97a].

But the lion's share of the middle end of the compiler was still implemented in an untyped intermediate language called CPS. This includes most optimization phases as well as the closure conversion. The FLINT project aims to use strongly typed languages for all phases of the compiler, so we needed to port the untyped optimizations out of the CPS part of the code to the earlier strongly typed phase. This included a minor redesign of the typed intermediate language to make it more amenable to optimization.

The optimizer in SML/NJ is fairly unsophisticated and mostly aims at cleaning up the most glaring inefficiencies introduced by the compiler or the language. My reimplementation does not attempt to go any further. Its main goal was to reduce the amount of work spent on untyped code. Of course, in order for the new optimizer to be fit for integration in the official SML/NJ distribution, I had to ensure that it did not slow down compilation or execution. To this end, the new code mostly does the same as the old one.

In the rest of this chapter, I will present the structure of the old middle end in Sec. 2.1, the new FLINT intermediate language in Sec. 2.2, how the middle end was changed in Sec. 2.3, then in Sec. 2.4 I discuss our experience comparing the A-normal form used in FLINT to the CPS form used in the old optimizer. Traditional untyped optimizations should ideally be unaffected by the addition of types; I discuss my experience in this respect in Sec. 2.5. finally, I will conclude with summary in Sec 2.7.

2.1 The old middle end

The SML/NJ compiler is composed of a front end, a middle end, and a back end. The front end does the lexical analysis, syntactic analysis, type inference, pattern matching compilation, as well as module language compilation. The back end takes care of the code generation, including register allocation and instruction scheduling. The middle end, which is the part of interest in this chapter, includes the optimization phases, the choice of representation of data-structures, and the closure conversion. A somewhat dated but still relevant presentation of the whole compiler can be found in [App91].

Prior to my work, the middle end was structured as follows [SA95]:

- PLambda: the typed language generated by the front end. This is basically a variant of the F_ω λ-calculus, extended with tuples and tagged sums. Functions take a single argument and return a single value. Every variable name is distinct, so there is never any possibility of name capture. This phase does A-normalization [FSDF93], which decomposes expressions into a sequence of elementary operations, where each intermediate value is bound to a variable.

- Lambda: the main type intermediate language. This language uses A-normal form but is otherwise the same as PLambda. This phase implements the representation analysis algorithm [Sha97a]. It is composed of the following phases:

- lcontract: this is a simple cleanup phase that inlines called-once functions to their sole calling location and flattens the let bindings by applying the let-associativity rule: $\mathsf{let}\ x = \mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\ \mathsf{in}\ e_3 \Rightarrow \mathsf{let}\ y = e_1\ \mathsf{in}\ \mathsf{let}\ x = e_2\ \mathsf{in}\ e_3$
- specialize: the type inference algorithm for SML is designed to find the most general type of expressions. This is good for programming, but regarding efficiency this often introduces unnecessary generality, so this phase instead specializes expressions to their least general type.
- wrap: this phase implements the core of the representation analysis [Sha97a], deciding which value needs to be boxed, which one needs to use coercions [Ler92] and which one type passing [HM95]. It also introduces the coercions where necessary.
- reify: this phase compiles away the type passing where it is used. In other words, it turns types into runtime data wherever this is needed. The output of this phase is not strongly typed any more, although it still has type annotations.
- convert: the last phase compiles switch statements to table-based computed jumps, turns the tags of tagged sums into plain integers, and most importantly converts the code to continuation passing style.

• CPS: the untyped intermediate language. This language is continuation based: functions never return and every primitive operation binds its return value to a variable. This phase first passes the code through a series of optimization phases, then applies closure conversion [SA94] and finally passes the result to the back end. The optimization phases are the following:

  - contract: dead code elimination, constant propagation, constant folding, inlining of functions only called from a single spot [AJ97]. This phase is executed after almost every other phase, as a form of cleanup and it can also be executed repeatedly because its output might still contain missed opportunities for simplification.

– etasplit: introduces η-redexes such that all functions fall into one of two categories: either all of their call sites are known or none of their call sites are known. I.e. if a function f is not in one of those two categories, a new function $f' = \lambda x.f\,x$ is introduced and all places where f escapes (i.e. is passed as a higher function argument rather than being directly called) are redirected to use f'.

– eta: elimination of η-redexes: replaces all expressions of the form $\lambda x.f\,x$ with f. Because this tends to undo the work of etasplit and because it is rarely beneficial, this phase is used only at the very beginning to cleanup the output of the CPS conversion, and at the end when η-redexes are not beneficial any more.

– expand: general inlining, which decides whether or not to inline functions called more than once based on a budget and on estimates of code size and of optimization opportunities that inlining will create. This also does loop unrolling, and introduction of loop pre-headers [App94] which allows loops to be inlined.

– flatten: argument flattening. As mentioned earlier, the earlier phases of the compiler use languages where functions take and return single values: when multiple values are needed, they are bundled in a tuple. This phase optimizes the code so that such bundles are opened up and passed directly in registers as multiple arguments or multiple return values.

– uncurry: simple form of uncurrying which turns curried functions fun $f\,x\,y = e$ into an uncurried function fun $f'(x, y) = e$ and an uncurry wrapper fun $f\,x\,y = f'(x, y)$. The actual uncurrying at the call sites is done later on in the *expand* phase by inlining the wrapper.

Additionally to the above, there had been some cross-module inlining code [BA97] implemented in a previous untyped version of Lambda and which was never integrated into the official distribution. My work involved reimplementing the optimizations performed in CPS as well as the cross-module inlining to the successor of Lambda, called FLINT.

```
datatype exp
  = RET of value list
  | LET of var list * exp * exp
  | FIX of fundec list * exp
  | APP of value * value list
  | TFN of tfundec * exp
  | TAPP of value * ty list
  | CON of con * ty list * value * var * exp
  | SWITCH of value * (con * exp) list * exp option
  | RECORD of value list * var * exp
  | SELECT of value * int * var * exp
  | RAISE of value * ty list
  | HANDLE of exp * value
  | PRIMOP of primop * value list * var * exp
withtype fundec = fkind * var * (var * ty) list * exp
     and tfundec = var * (tvar * kind) list * exp
```

Figure 2.1: The FLINT datatype

2.2 The FLINT intermediate language

The new typed intermediate language, called FLINT, is a minor update of the Lambda intermediate language, intended to make it more appropriate for optimization. Other than cosmetic changes, the only real difference is the fact that functions now take multiple arguments and return multiple values.

It is a variant of the F_ω λ-calculus using an A-normal representation, which means that every intermediate value is bound to a variable. More concretely, it means that instead of providing a primitive expression like $(v_1 + v_2)$, the language always uses expressions of the form $\mathsf{let}\ x = v_1 + v_2\ \mathsf{in}\ e$, so $(v_1 + (v_2 + v_3))$ becomes $\mathsf{let}\ x_1 = v_2 + v_3\ \mathsf{in}\ \mathsf{let}\ x_2 = v_1 + x_1\ \mathsf{in}\ e$. Figure 2.1 shows a slightly simplified definition of the SML datatype used to represent FLINT expressions. `value` can be a variable `var` or an immediate integer. We decided to keep type annotations to a minimum, so they are only added where a simple type propagation traversal cannot infer them: mostly, function arguments.

Expressions evaluate to a list of values, returned with `RET` $\vec{v}$. `LET`$(\vec{x}, e_1, e_2)$ executes e_1, binds the results to $\vec{x}$, and then executes e_2.

FIX($\overrightarrow{fun}, e$) defines a set of mutually recursive functions bound in the scope of e. Each function has a name (a variable to which it is bound), a list of arguments with their type and a body. The variable to which the function is bound has type $\vec{\sigma_1} \to \vec{\sigma_2}$ where $\vec{\sigma_1}$ is the list of types of the function's arguments and $\vec{\sigma_2}$ is the list of types of the function's return values. For non-recursive functions, $\vec{\sigma_2}$ is inferred when needed by type reconstruction, whereas for recursive functions $\vec{\sigma_2}$ is recorded in the `fkind` part of the `fundec`. APP($v, \vec{v}$) applies a function v to a list of arguments $\vec{v}$.

TFN($tfun, e$) defines a type function *tfun* bound in the scope of e. A type function has a name (a variable to which it is bound), a list of type parameters with their kind, and a body. The type of the variable to which the type function is bound is $\forall \overrightarrow{t:\kappa}.\vec{\sigma}$ where $\overrightarrow{t:\kappa}$ is the list of type arguments with their kind and $\vec{\sigma}$ is the type of the return results. Type functions are used to implement polymorphism. TAPP($f, \vec{\sigma}$) passes the type arguments $\vec{\sigma}$ to the type function f. Type functions are side-effect free, so that any TAPP can freely be moved, copied, or deleted.

CON($con, \vec{\sigma}, v, x, e$) constructs a tagged value of constructor *con* and carrying the value v. The result is then bound to x while executing e. $\vec{\sigma}$ is the list of types used to instantiate the constructor in cases when it is polymorphic. CON is used when constructing datatype elements. For example, when building an option value SOME v, *con* is SOME and $\vec{\sigma}$ is the type of v. SWITCH($v, \overrightarrow{(con, e)}, e_{\mathsf{def}}$) jumps to the e whose *con* matches the tag of v, or executes e_{def} if none of the branches match. Additionally to jumping, SWITCH also extracts the value v carried by the constructor *con* and binds it to a free variable of e.

RECORD($\vec{v}, x, e$) makes a tuple carrying values $\vec{v}$, binds it to x, and executes e. Just as in SML, tuples are not mutable. SELECT(v, i, x, e) extracts the i^{th} element of tuple v, binds it to x and executes e.

RAISE($v, \vec{\sigma}$) raises an exception v. $\vec{\sigma}$ is the type of the expression: since the expression does not return, we cannot infer its type otherwise. HANDLE(e, v) executes e while the function v is temporarily used as the exception handler.

PRIMOP($po, \vec{v}, x, e$) executes the primitive *po* with arguments $\vec{v}$, binds the single result to x and then executes e. Primops include, arithmetic operations, ref-cell operations, array operations, and a lot more.

In order to avoid problems of name capture, every variable name is unique and this invariant is preserved by α renaming whenever a piece of code is duplicated.

2.3 The FLINT optimizer

The first part of the work was to port the old Lambda phases to the new FLINT representation. I will not discuss this for two reason: first because it was straightforward, given the similarity between the two languages, and second because most of the work was done by other people.

As explained in the introduction, since the purpose was not to do better than the old optimizer but to do just as good, the new optimizer basically applies the same optimizations as the old one. Nevertheless, the two are structured fairly differently and because the languages on which they operate are different, the optimizations they perform are not always the same.

The main differences are the following:

- Flattening: instead of the untyped local flattening optimization performed by the old optimizer, the new optimizer uses a type-directed flattening that is built on top of the pre-existing representation analysis phase.

- *known* functions: the old optimizer restricted several optimizations to known functions only, i.e. functions whose call sites are all statically known. The new optimizer uses optimizations that work either way instead.

- General organization: the phases in which specific optimizations are done are changed in order to try to reduce the number of traversals.

These points are explained below.

2.3.1 Known functions

The old optimizer conditionalized several optimizations on whether or not the function that is a candidate for an optimization is *known*, i.e. all its call sites are statically known. It was done when the optimization needed to change not only the function but also all its calls. This was the case for example for flattening or for elimination of dead arguments. To counteract this, the etasplit phase was introducing η-redexes for every function whose call sites were only partly known, such that all the known call sites would call the original function while the other places where the function was used referred to the η-redex instead.

In the new optimizer, I simplified those optimizations so that they do not need to update the call sites at all. Instead, when an optimization such as removal or dead arguments is performed, it splits the function into a small inlinable wrapper that uses the same calling convention as before and the main body whose calling convention is optimized. This technique was already used for uncurrying and I simply applied it wherever possible. The advantage is that it eliminates the need for the distinction between known and unknown functions, allows the optimization to be done completely locally without having to pay attention to the way the function is used, and lets the generic inliner do the actual work of updating the callers.

2.3.2 General organization

The FLINT optimizer is made of the following phases:

- flintnm: translate PLambda into FLINT. Other than cosmetic changes between the two languages, this is really a combination of A-normalization and the first phase of the flattening.

- fcontract: elimination of dead code, constant propagation, constant folding, function inlining, η-reduction, flattening of let bindings using the let-associativity rule. This closely corresponds to the combination of the CPS contract and eta phases, except that it also does general inlining, which used to be done in expand.

- lcontract: this phase was carried over from Lambda and does a subset of what fcontract does. It does a much less thorough job, but is much faster and was kept to do the first cleanup after translation from PLambda.

- loopify: look for functions that call themselves, wrap them in a pre-header, eliminate arguments that stay constant throughout the loop, and check whether all the recursive calls are in tail position, in which case the loop is marked as being a *while* loop. The corresponding optimization in the old optimizer was done in expand.

- fixfix: decide which functions are candidates for inlining, rewrite curried function like uncurry did in the old optimizer, and break up groups of apparently mutually-recursive functions into smaller subgroups. Other than introducing loop pre-headers, done in loopify, and deciding which function calls to inline and to perform the inlining itself, done in fcontract, this phase corresponds to the expand phase of the old optimizer.

- split: split top-level functions corresponding to SML functors into a small inlinable component and a large component containing the rest. The inlinable component is then added to the compilation units that refer to the current one, for cross-module inlining. This phase does not correspond to any optimization performed by the CPS optimizer, but corresponds instead to the lsplit phase that had been implemented in an earlier untyped incarnation of Lambda [BA97].

Part of the reason for having placed the different optimizations in different phases is just simplicity or expediency, but an important aspect was compile time. There are several reasons why compile time tends to be higher with a FLINT-based optimizer than with a CPS-based one, but it seems that the most important ones are the fact that the CPS terms have had their `TAPP` and `TFN` terms removed, the fact that a CPS representation is a bit more uniform, so the code of the optimizer is a bit simpler (see the Sec. 2.4 for a discussion of these issues), and the fact that the FLINT optimizer operates earlier in the compiler, so we still need to perform a bit of optimization in CPS after the FLINT optimizer in order to clean up some inefficiencies introduced by later phases.

The specific reasons for the new structure are as follows:

- As explained before, the need for etasplit is avoided by using slightly different formulations of the relevant optimizations.
- It was easy and harmless to do uncurrying as part of fixfix, with the added benefit that it merged two tree traversals into one. This is particularly important since those phases are worth doing often and both traversals need to build a whole new FLINT expression, which is slower than read-only traversals.
- It seemed easy and harmless to move η-elimination into fcontract, and with similar benefits as above. In retrospect, it took a long time to debug, which, I later learned, was the main reason why it was a separate phase in CPS.
- The reason to move the actual inlining out of fixfix and into fcontract was so that cascading inlining could take place. It also simplified fixfix slightly. Another reason was that split requires the choice of inlining candidates to be separate from the inlining itself.
- loopify was moved out of fixfix because it does not need to be run as often, but it requires two passes (a first pass that collects information and a second that does the code transformation), whereas fixfix is implemented in a single pass.

2.3.3 Flattening

All SML functions take a single argument and return a single value. In order to pass multiple arguments to a function, you can either use currying or you can combine the arguments into a tuple or a record. The SML syntax makes tupling and untupling mostly transparent. In the example below, `(a,b)` is a pattern that matches a tuple and extracts its elements and `(b,a)` builds a new tuple:

```
fun f (a,b) = g (b,a)
```

This use of tuples is very convenient for the programmer, but for performance reasons the compiler needs to recognize such uses and eliminate the construction/deconstruction of tuples by passing the multiple values in separate registers instead.

This *flattening* optimization used to be performed in the old optimizer by a phase that would look for functions taking a tuple as argument and if all the call sites were statically known, it would change the function and all its callers. In the new optimizer we decided instead to take advantage of the representation analysis to do a type-directed flattening. If a function's argument is of tuple-type, the function is changed to take the various elements of the tuple as separate arguments. Function calls are also correspondingly updated.

The main advantage of such a type-directed approach is that it works even if some of the call sites are not statically known. More specifically, this means that flattening takes place even if the function is exported to or imported from another compilation unit, or even if it passed as an argument to another function. Incidentally, in CPS, continuation functions are always passed to other functions, which means that return values were not flattened, whereas our approach does flatten return values just like it flattens arguments.

The presence of polymorphism makes type-directed flattening more complex. Let us take the identity function:

```
fun f x = x
```

clearly this function cannot be flattened. Its type is $\forall t.t \rightarrow t$: it takes an argument of any type t and returns a value of the same type. If we instantiate t to $\mathsf{int} \times \mathsf{int}$ we can use this function anywhere where a function of type $\mathsf{int} \times \mathsf{int} \rightarrow \mathsf{int} \times \mathsf{int}$ is expected. But such calls necessarily have their arguments and return values flattened. For this reason, when we instantiate t to $\mathsf{int} \times \mathsf{int}$, we need to add coercion wrappers that adapt the calling convention of the polymorphic code to the calling convention corresponding to its instantiation. Figure 2.2 shows an example where the flattening phase had to complement the `TAPP` with a coercion wrapper so that f_3 gets the type expected by the rest of the code. Those wrappers of course get inlined wherever possible. The insertion of those coercion wrappers is taken

```
let fun f x = x
in ... f (x:int*int)
```

$\Downarrow$ unflattened translation $\Downarrow$

$$
\begin{array}{l}
\mathtt{TFN}\ f_1\ [t\!:\!\Omega] = \mathtt{FIX}\ f_2\ [x_1\!:\!t] = \mathtt{RET}[x_1] \\
\qquad\qquad\qquad \mathtt{IN}\ \mathtt{RET}[f_2] \\
\mathtt{LET}\ [f_3] = \mathtt{TAPP}(f_1, [\mathsf{int} \times \mathsf{int}]) \\
\quad \vdots \\
\mathtt{IN}\ \mathtt{APP}(f_3, [x_{15}]))))
\end{array}
$$

$\Downarrow$ flattening $\Downarrow$

$$
\begin{array}{l}
\mathtt{TFN}\ f_1\ [t\!:\!\Omega] = \mathtt{FIX}\ f_2\ [x_1\!:\!t] = \mathtt{RET}[x_1] \\
\qquad\qquad\qquad \mathtt{IN}\ \mathtt{RET}[f_2] \\
\mathtt{LET}\ [f_4] = \mathtt{TAPP}(f_1, [\mathsf{int} \times \mathsf{int}]) \\
\mathtt{FIX}\ f_3\ [x_2\!:\!\mathsf{int}, x_3\!:\!\mathsf{int}] = \\
\qquad \mathtt{LET}\ x_4 = \mathtt{RECORD}[x_2, x_3] \\
\qquad \mathtt{LET}\ [x_5] = \mathtt{APP}(f_4, [x_4]) \\
\qquad \mathtt{LET}\ x_6 = \mathtt{SELECT}(x_5, 0) \\
\qquad \mathtt{LET}\ x_7 = \mathtt{SELECT}(x_5, 1) \\
\qquad \mathtt{IN}\ \mathtt{RET}[x_6, x_7] \\
\quad \vdots \\
\mathtt{LET}\ x_{16} = \mathtt{SELECT}(x_{15}, 0) \\
\mathtt{LET}\ x_{17} = \mathtt{SELECT}(x_{15}, 1) \\
\mathtt{IN}\ \mathtt{APP}(f_3, [x_{16}, x_{17}]))))
\end{array}
$$

Figure 2.2: Flattening polymorphic code.

care of by the representation analysis code which was already doing the same kind of things in order to box/unbox data structures.

As can be seen from the example, flattening can introduce a lot of extra code around type applications, in the form of wrappers that then require a lot of cleanup. Furthermore, we cannot not always inline the wrappers at the call sites and we are often unable to "inline" the wrappers into the wrapped function, so wrappers are difficult to eliminate once introduced, which means that we should avoid introducing such wrappers before we know that they will be needed. In practice this means that we should not flatten before specialize since specialize will eliminate a large number of type applications.

But at the same time, flattening can enable further optimizations, so we wanted to perform flattening early. So we decided to have our cake and eat it too: flattening is done in the first phase (flintnm), but the coercion wrappers are only inserted later, after specialize in the wrap phase which was already taking care of other coercion wrappers. In other words, until we get to executed wrap, our type applications are implicitly doing the coercions, such that in the previous example, TAPP(f_1, [int × int] magically returns a flattened function even though f_2 is not flattened.

This has two side effects: first in specialize, we have to flatten functions whose argument or return type we change from a single polymorphic argument to a tuple. This turns out to be very good since it means that we flatten polymorphic code as early as possible. Second, the type system becomes more complex: when we instantiate a type like $\forall t.[t] \rightarrow [t]$ to int × int, we need to get $\forall t.[\mathsf{int}, \mathsf{int}] \rightarrow [\mathsf{int}, \mathsf{int}]$ instead of $\forall t.[\mathsf{int} \times \mathsf{int}] \rightarrow [\mathsf{int} \times \mathsf{int}]$, which means that the function we use to do type instantiation cannot just substitute int × int for t, but needs to flatten the resulting type if necessary.

2.3.4 Contracting β-redexes

The fcontract phase is really the workhorse of the FLINT optimizer, even more so than contract was for the old optimizer. The reason is that most other optimizations limit themselves to detecting and enabling optimization opportunities while leaving the actual work to fcontract. So it needs to do a thorough job when performing those optimizations and experience shows that it is easy to write a contraction phase that leaves a lot of optimization opportunities in its output, requiring repeated execution to get a good result.

A contraction phase is generally structured as a first phase that collects usage info to determine liveness of variables as well as to figure out which functions are only called once, and a second phase that performs the contractions. The problem is that contractions tend to cascade such that after having performed one contraction, others become possible, but the counters might not reflect it. For example, when eliminating a dead function, some other function might become dead or might have its call-count reach one, but unless the counters are properly updated, the optimization will be missed.

To minimize this problem, fcontract uses the same approach as was used in contract [AJ97]: counters are updated as optimizations are performed. Actually, fcontract is a bit more aggressive in that the counters of variables referred to by a term are decremented as soon as the term becomes dead, whereas in contract the decrement was only taking place on the way up of the recursion.

An important difference between the old contract and the new fcontract is the fact that fcontract performs general inlining rather than only inlining called-once functions. As mentioned, this allows cascading inlining. A typical example of cascading inlining is when `map` is passed an inlinable function: only after inlining `map` can the function argument be inlined. But this also runs the risk of inlining indefinitely.

Preventing infinite inlining is a classical problem and it is solved very straightforwardly in fcontract by keeping track of the stack of functions we are currently inlining so as to detect and break inlining cycles. The first attempt at solving the problem was to decide that recursive functions could not be inlined, but it had two disadvantages: first, it is not enough because you can use a recursive datatype to create a loop without any recursive function, second it is too restrictive because many wrappers such as uncurry wrappers can be recursive and yet should be inlined.

2.4 A-normal vs CPS

Other than the fact that the new optimizer works on typed terms, it is also different from the old optimizer in that it works on A-normal terms rather than CPS terms. In [FSDF93], A-normalization was proposed as an alternative to CPS that provides the same benefits but while preserving more of the source-level structure. In this section, I will discuss the various places in which the difference between the two seemed relevant.

2.4.1 The difference

In general, a term is said to be in CPS form if all the function calls are tail calls. That means that non-tail function calls need to be modified by passing an explicit *continuation* (which is equivalent in some sense to a return address and an activation frame), which the called function will then call when it wants to return. Also, all primitive operations only take immediate values or variables as arguments, rather than expressions, and bind their result to a variable, so every intermediate value has a name and all operations are explicitly sequentialized. Since tail-calls are only a step away from an assembly `jump` instruction, the CPS representation of a program provides a nice mix of being very close to assembly code while still enjoying the high-level formalism provided by the λ-calculus.

A-normal form keeps the requirement that all primitives only take an immediate value or a variable as argument and that they bind their result to a variable, so that primitive operations are sequentialized and all intermediate values are named, but it allows non-tail function calls. The idea behind it is that, this way, the main structure of the code is kept and we do not need to introduce explicit continuations. One of the advantages of keeping continuations implicit, is that it constrains the control flow in such a way that activation frames can be allocated on a stack. This advantage is not really interesting in our case, since the rest of the SML/NJ system uses heap-allocated activation frames anyway.

In terms of the definition of allowed terms, the difference between CPS and A-normal form is fairly small, and to a first approximation, we could say that the main difference between the two is embodied in FLINT's `LET` terms which CPS would disallow. For this reason, it is interesting to look at the kinds of codes in FLINT that use `LET`. Most uses of `LET` are removed by simple local rewriting, the rest falls into the following 4 cases:

$$\begin{array}{l}
\texttt{LET}\ \vec{x} = \texttt{APP}(v, \vec{v})\ \texttt{IN}\ e \\
\texttt{LET}\ \vec{x} = \texttt{TAPP}(v, \vec{\sigma})\ \texttt{IN}\ e \\
\texttt{LET}\ \vec{x} = \texttt{HANDLE}(e_1, v)\ \texttt{IN}\ e \\
\texttt{LET}\ \vec{x} = \texttt{SWITCH}(v, \overrightarrow{(con, e)}, e_{\mathsf{def}})\ \texttt{IN}\ e'
\end{array}$$

Strictly speaking, FLINT is not using A-normal form, because the A-normal form was defined such that the fourth case was disallowed as well. More specifically, it would have been turned into something of the form:

$$
\begin{array}{rcl}
\texttt{SWITCH}(v, con_1 & \Longrightarrow & \texttt{LET } \vec{x} = e_1 \texttt{ IN } e' \\
con_2 & \Longrightarrow & \texttt{LET } \vec{x} = e_2 \texttt{ IN } e' \\
\ldots & & \\
con_n & \Longrightarrow & \texttt{LET } \vec{x} = e_n \texttt{ IN } e' \\
\mathsf{def} & \Longrightarrow & \texttt{LET } \vec{x} = e_{\mathsf{def}} \texttt{ IN } e')
\end{array}
$$

Of course, such a transformation is naive since it blindly duplicates e' and will lead to uncontrolled code size explosion. An easy way to address this issue is to use the following instead:

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\texttt{FIX } k\ \overrightarrow{x\!:\!\sigma} = e'} \\
\texttt{SWITCH}(v, con_1 & \Longrightarrow & \texttt{LET } \vec{x} = e_1 \texttt{ IN APP}(k, \vec{x}) \\
con_2 & \Longrightarrow & \texttt{LET } \vec{x} = e_2 \texttt{ IN APP}(k, \vec{x}) \\
\ldots & & \\
con_n & \Longrightarrow & \texttt{LET } \vec{x} = e_n \texttt{ IN APP}(k, \vec{x}) \\
\mathsf{def} & \Longrightarrow & \texttt{LET } \vec{x} = e_{\mathsf{def}} \texttt{ IN APP}(k, \vec{x}))
\end{array}
$$

The MLJ compiler [BKR98] used something similar, except that they additionally constrained their `SWITCH` statement such that each branch is just a function call. We decided not to stick to a strict A-normal form because it is not closed under β-reduction. In other words, if we replace an `APP` term by inlining the function, the resulting code might not be in strict A-normal form, so unless we are careful to always re-normalize the code simultaneously to inlining, we have to allow at least temporarily a de-normalized representation. Neither CPS nor our loose A-normal representation suffer from such a problem.

2.4.2 Advantages of using CPS

When compared to the original optimizer, the FLINT optimizer had several new problems to deal with. Some of them come from the difference between CPS and A-normal form, while others come from the difference between FLINT and strict A-normal form. The most obvious advantage of CPS is that returning from a function is done by a function call, so all optimizations written for function applications magically apply to function returns as well. A lot of optimizations that came for free in CPS have to be explicitly handled in FLINT, most notably:

- Inlining called-once functions in CPS subsumes the let-associativity rules in FLINT. Let-associativity is used by fcontract and lcontract to normalize code by replacing the various forms of

```
let x1 = let x2 = e1 in e2 in e3
```

with

```
let x2 = e1 in let x1 = e2 in e3
```

- Elimination of η-redexes in CPS subsumes the elimination of `LET` $\vec{x} = e$ `IN RET` $\vec{x}$ in FLINT.

- Elimination of dead function arguments in CPS subsumes elimination of dead variables in `LET` bindings in FLINT. Incidentally, this optimization is not fully implemented in FLINT because it requires a lot of extra processing.

- Removal of invariant function arguments in CPS subsumes the special code that detects tail-recursive loops, since it is nothing more than hoisting the continuation argument out of the loop. Also, the lack of `GOTO` in FLINT means that tail-recursive loops need to be marked with an ad-hoc annotation, whereas the corresponding optimization in CPS is all done within the language with no need for special annotations.

- Exception handling can be compiled away in CPS into plain functions and function calls, at which point function inlining subsumes elimination of `RAISE` inside a `HANDLE`, dead function elimination subsumes dead handler elimination, and so on. The old CPS language does not compile exception handling away, so it does not take advantage of this.

- Function inlining in CPS subsumes path-splitting in FLINT. Path-splitting is an optimization that turns

```
let x = (if a then b else c) in e
```

into

```
if a then (let x = b in e) else (let x = c in e)
```

> The FLINT optimizer does not perform this optimization because it did not seem worth the effort.

There are other circumstances where a CPS representation is more beneficial. The most obvious one is for exception handling: the restricted control flow in A-normal form makes any kind of optimization of exception handling very difficult. Luckily the old optimizer performed only very limited optimization of exception handling, so the lack of any such optimization in the new optimizer does not penalize it too excessively.

Another case where a CPS representation is helpful is with tail recursive loop exits, as in the example below:

```
fun loop x = if e1                      let k y = e
             then e2            CPS     let loop x = if e1
             else loop e3       ==>                  then k e2
let y = loop v                                       else loop e3
in e                                    in loop v
```

The direct-style version on the left does not make the connection between `e2` and `e` explicit, whereas the CPS transformation shows quite clearly that `k` can be inlined to move `e` into the exit of the loop. The FLINT optimizer does not attempt to recognize such cases.

2.4.3 Advantages of A-normal

Of course, an A-normal representation of code has its own advantages. One already mentioned is that it puts the necessary constraints on control flow to ensure that the activation frame obey a stack discipline.

Another one is that in CPS, the return continuation is just a higher-order function and there is no obvious indication of the fact that its use follows a special pattern: for example, if all the call sites of a function are known, then its continuation argument can only contain values within a particular set and those continuations will not be used anywhere else than in this function.

Such constraints ensure that we can easily change the calling convention between this call site and the corresponding functions. Optimizations that could take advantage of this are optimizations such as flattening of return values, or elimination of dead or constant return values. Indeed the flattening phase in CPS did not flatten return values and it would have been difficult to extend it to do that, whereas a translation of that phase into FLINT could have been naturally extended to also flatten return values. As it turns out, we used a type-directed flattening algorithm, which is oblivious to such issues and would work just as well in a CPS representation.

2.5 Impact of types

While porting those classical untyped optimizations from the untyped CPS language to the typed FLINT, most of the work was unaffected by the addition of types. But there were of course some exceptions. Types interact with the optimizer in several ways:

- Too many type annotations can make the compiler writer's life more difficult by having to do a lot of work to maintain those types through the various phases. They can also lead to a slowdown of the compiler.

- Too few types annotations can make some optimizations difficult. This can happen when the optimization's output requires more type annotations than the input, in which case some type inference is needed.

- Additional terms that manipulate types such as `TAPP` or `TFN` make the optimizer more complex and potentially slower, but more importantly they can hide or prevent optimizations.

- Types can provide additional information which can enable more optimizations. This is rather rare in the context of traditional untyped optimizations.

Here are in no particular order the concrete ways in which types influenced the FLINT optimizer:

- Manipulating types can be costly. The Lambda intermediate language used a very efficient type representation which was reused for the FLINT language [SLM98], so we had already addressed most of the performance trouble spots. But we did encounter cases where we used types naively in the flattening code, which lead to an exponential complexity in some cases. The problem was that we traversed our types as if they were trees, instead of being careful to traverse them as graphs to benefit from the large mount of sharing.

- `LET` $[x] =$ `RAISE`$(y, \vec{\sigma})$ `IN` e: in code such as this, e is clearly dead and we should be able to replace the whole expression with `RAISE`$(y, x\vec{\sigma'})$. The problem here is that $\vec{\sigma'}$ is the type of the values returned by e and this type is not readily available. Because of this problem, the FLINT optimizer does not perform dead code elimination for the above code. There are many ways to solve this problem, from removing the type annotation on `RAISE` to adding type annotations at other places, but my favorite solution is probably to use a CPS representation of the code, in which case this dead code elimination comes for free.

- `LET` $x_0 =$ `SELECT`$(x, 0)$ ... `LET` $x_n =$ `SELECT`(x, n) `IN RECORD`$(x_0, ..., x_n)$: to replace this record construction with just a reference to x we need to ensure that x is of size n. This only works if we have the corresponding type information. The FLINT optimizer does perform this optimization when the type information is easily accessible. It is not often useful, but it can help when flattening over-enthusiastically. This optimization might also be considered valid even if the size of x is not n (i.e. even with no type information), but it can lead to memory leaks.

- The original version of FLINT used DeBruijn indices in its representation of types. This meant that moving code into or out of a `TFN` term required adjusting all the embedded type annotations. This was sufficiently tricky that I decided at first to temporarily disallow function inlining into a `TFN`. Later Christopher League changed the type representation to use named variables for type variables bound by `TFN`, while still keeping DeBruijn indices for type variables bound inside the types themselves. Thanks to this, the FLINT optimizer now ignores DeBruijn indices and inlines into

TFNterms.

- Inlining TFN terms is difficult: there are several problems at play here. First is the fact that the classical inlining heuristics are designed for functions that take actual values as arguments and use them actively in their body. Those heuristics work poorly for functions of no arguments, which is how TFN would be treated. But additionally to this, in the case of FLINT, inlining a TFN is made more difficult by the fact that we may need to do flattening since we are instantiating the type variable. Because of this, fcontract never inlines TFN. On another hand, specialize does properly eliminate any TFN that is called only once.

- TFN hides its content, preventing other optimizations, such as in the following case:

$$
\begin{array}{l}
\texttt{TFN}\ f\ [t\!:\!\Omega] = \texttt{FIX}\ g\ [x\!:\!\sigma_x] = \ldots \\
\qquad\qquad\qquad\ \ \texttt{FIX}\ h\ [y\!:\!\sigma_y] = \ldots \\
\qquad\qquad\qquad\ \ \texttt{IN RET}\ [g, h] \\
\ldots \\
\texttt{LET}\ [g, h] = \texttt{TAPP}(f, [\sigma]) \\
\texttt{IN}\ \ldots \texttt{APP}(g, \vec{v}) \ldots
\end{array}
$$

  In an untyped setting, both g and h would be defined at top level rather than inside f, so if g is a small inlinable function, it gets inlined as a matter of course. But the presence of TFN hides this and the FLINT optimizer will miss this opportunity. Maybe the λ-splitting code in split could be reused to solve those cases.

- Similarly, a polymorphic curried function will sometimes not be uncurried because the curry wrapper is hidden in the outside TFN used to represent polymorphism. The FLINT optimizer does not currently attempt to handle this case either and to uncurry the TFN together with the function.

2.6 Related work

Most functional language compilers rely on similar optimizations as presented here. But few of them use typed intermediate representations.

The TIL project [TMC+96] is probably the closest. They also use an intermediate representation derived from A-normal form and they implement the same untyped optimizations, plus a few loop-based optimizations such as array bounds check elimination, common subexpression elimination, and loop invariant removal. Their flattening is not type-directed.

The MLJ compiler [BKR98] also uses a similar setup to ours. Its internal types include some effect annotations which allows it to do type-directed uncurrying and its backend is quite different because of the need to adapt to the idiosyncrasies of JVM, but the untyped optimizations are otherwise very similar to ours.

The Glasgow Haskell compiler [PMar, PS98] also uses a typed intermediate language and a very similar structure of the optimizer. It does not use type-directed optimizations for flattening, data-representation, or uncurrying but still carries types to implement Haskell's type classes dictionaries.

2.7 Conclusion

The port of the traditional optimization from the untyped CPS intermediate language to the typed FLINT intermediate language did not suffer excessively from the fact that the target language was using A-normal form or that it was strongly typed, although both aspects turned out to be minor impediments.

In retrospect, a language based on a CPS representation of the code is better suited to those kinds of optimizations because it is more uniform and makes several things more explicit, most importantly control flow, which would help optimize tail-recursive loops exits, exception handling, and sequences of conditionals. Of course, a CPS representation might introduce other problems if we had to use a stack-discipline for our activation frames. It seems that a technique like the one presented in [DL92] could be used to recover the needed information, but I can only speculate.

Also, the trick of spreading flattening from `flintnm` to `wrap` through several other phases is probably not worth the effort. It seems like it would be better to do it once and for all in `wrap` so the other phases can manipulate the code more freely.

The new optimizer has been included in the official distribution of SML/NJ since version 110.26. Interestingly enough it was first integrated in version 110.10, then removed because of a bad interaction with some other change. It took a year of debugging and finger pointing to finally fix the bug which happened to be caused by generated code that did not obey the garbage collector's expected interface. This would not have happened had SML/NJ used a formalized and automatically checked interface between its collector and mutator, like the ones I will show in the second half of this thesis.

Chapter 3

Inlining as staged computation

Clear and maintainable code requires modularity and abstraction to enforce well-designed interfaces between software components. The module language of Standard ML [MTHM97] provides powerful tools for such high-level code structuring. But these constructs often incur a considerable performance penalty which forces the programmer to break abstraction boundaries or to think twice before using advanced features like parameterized modules (e.g., ML functors).

Efficient implementation of these high-level language constructs often rely crucially on function inlining. Inlining algorithms have been used for many years, but their "best-effort" behavior prevents me from knowing or making sure that a function will always be inlined (at least, wherever possible given the compilation model). For example, SML/NJ has several ad-hoc tricks sprinkled in the code to expand primitive operations. These tricks tend to muddy up the abstraction boundaries so it would be nice if they could be replaced by a general-purpose inlining algorithm.

But would the inliner perform as good a job inlining those primitive operations as with the ad-hoc approaches? For simple cases, it is straightforward to ensure that primitive operations are always inlined, but when higher-order functions or even higher-order modules (such as SML/NJ functors or Java generics) come into play, coupled with separate compilation, the question becomes more challenging. In the course of implementing the extension of Blume and Appel's cross-module inlining algorithm [BA97], we tried to understand the relationship between inlining opportunities and separate compilation. We felt a need to formalize the solution to better understand its behavior.

This chapter is the result of my efforts to formalize the inlining algorithm presented in the previous section. More specifically, I borrow from the partial-evaluation community [JGS93] to model inlining as a staged computation. By using a two-level λ-calculus [Mog97] as my intermediate language, I can assign each function (in the program) with a binding-time annotation of *static* or *dynamic*: a static function call is executed at compile time and thus is always inlined, while a dynamic call is executed at run time and thus is not inlined. The inlining optimization is then equivalent to the standard off-line partial evaluation: First, use the binding-time analysis to locate all the inlining candidates, then run the specialization phase to do the actual inlining. The binding-time attributes can also be exported to the source level (or the compiler front-end) to serve as inlining annotations and to allow programmers (or the compiler writer) to control various inlining decisions manually.

Apparently, all partial evaluators support some form of β-reductions as part of the specialization, however, these techniques do not immediately apply to the inlining optimization. Because of the different application domains, partial evaluators are generally much more aggressive than compiler optimizers. Even the binding time annotations can pose problems at the source level because they can clutter the module interface and interact badly with ML functors; for example, I would have to add abstraction over binding-time (commonly called "binding-time polymorphism") in the type if I wanted to apply a functor to modules with different binding time (but with same signature otherwise).

The main objective of this chapter is to hammer out these details and to see what it would take to launch various partial-evaluation techniques into real compilers. This chapter builds upon previous work on cross-module inlining [BA97, Sha98, Ler95] and two-level λ-calculus [Mog97, NN92, DP96, TS97] but makes the following new contributions:

- As far as I know, my work is the first comprehensive study on how to model inlining as staged computation. The formalism from the staging calculus allows me to explicitly reason about and manipulate the set of inlinable functions. Doing such reasoning is much harder with a traditional inlining algorithm, especially in the presence of ML-style parameterized modules.

- By careful engineering of binding-time coercions, and combined with proper staging and splitting, I show how to model inlining as staged computation without introducing unwanted code duplication (see Sec. 3.4).

- Adding inlining annotations to a surface language allows a programmer to mark a function as inlinable explicitly. Inlining annotations, however, could pollute the module interface with additional binding-time specifications. This makes the underlying module language more complex. I show how to support inlining annotations while still avoiding such pollution. In fact, my scheme is fully compatible with ML-style modules and requires no change to its module language.

- Using a two-level λ-calculus, I show how inlining annotations are compiled into the internal binding-time annotations and how they interact with the module code. This allows me to propagate inlining information across arbitrarily functorized code, even in the presence of separate compilation.

- I extend binding-time coercions to work with parametric polymorphism.

The rest of this chapter is organized as follows: Section 3.1 gives an overview of a compiler that supports cross-module inlining and shows how inlining annotations (at the source level) and two-level λ-calculus (as intermediate language) fit into the picture; Section 3.2 formally defines my source language SRC which supports inlining annotations and (indirectly) ML-style modules; Section 3.3 formally defines my target language TLC which is a two-level λ-calculus supporting staged computation; Section 3.4 presents my detailed algorithm for compiling SRC programs into TLC; the algorithm involves staging, splitting, and careful insertion of binding-time coercions; Section 3.5 shows how to handle top-level issues for

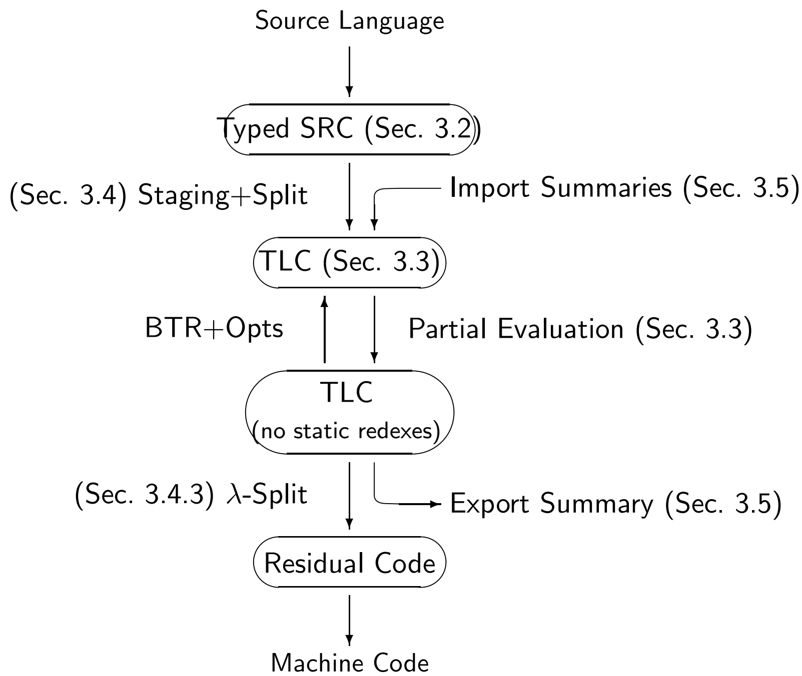


Figure 3.1: Structure of the compiler

inlining across multiple compilation units; Section 3.6 then presents several extensions over the basic algorithm; finally, Section 3.7 and 3.8 describe related work and then conclude.

3.1 The big picture

To model inlining as staged computation, we first give an overview of a compiler that supports cross-module inlining. We use our FLINT compiler [Sha97b, SLM98] as an example. Figure 3.1 shows various stages of compilation used in the compiler. The source code is first turned into a strongly typed intermediate language based on a predicative System-F calculus (we name it SRC and present its details in Sec. 3.2). The SRC calculus contains a module language and a core language. Each core-language function is annotated with inlining hints to indicate whether the function should be inlined or not. Those hints could be provided by the user or by the earlier phases of a compiler (using some inlining heuristics).

The inlining hints are then turned into staging annotations, mapping inlinable functions to static functions (functions executed at compile-time) and the rest to dynamic code (executed at run-time), by translating the code into a two-level intermediate language extended with polymorphism (we name it TLC and present its details in Sec. 3.3).

To minimize the performance cost of the module code, we want to mark it as static so as to expose as many inlining opportunities as possible. But this would imply that each functor application (SML's equivalent to template instantiation) would create a duplicate copy of the full functor body. This approach, while sometimes acceptable, can lead to excessive code growth and compilation times for heavily functorized code, as any programmer who has worked with C++ templates knows.

We use a variant of the λ-splitting technique [BA97] to split each module function into a static part and a dynamic part. This splitting is done carefully to ensure that it does not obfuscate any inlining opportunity. Splitting is done together with staging in the main translation algorithm (see Sec. 3.4.4). The resulting code is completed by incorporating a copy of the *summaries* from all the import modules (see Sec. 3.5); a *summary* contains the code that should be inlined across compilation-units, similarly to OCaml's *approximations* [Ler95].

The static part of the code is then reduced by a straightforward partial evaluation returning the same code but with no remaining static redexes. This code then goes through a binding-time refinement (BTR) or other optimization phases which could introduce new static code requiring a new pass through the partial evaluator.

Once these optimization steps are finished, we reuse the λ-splitting algorithm to split the compilation-unit itself into a *summary* containing all the remaining static code (i.e. inlinable code for future cross-compilation-unit inlining) and a fully dynamic residual code (encompassing the bulk of the program) which is then passed to the code generator.

Inlining across compilation units increases the coupling between those units. If a unit is modified, all units that import it will now need to be recompiled, even if the modification was only internal and did not change the interface. This is automatically handled in our case by a compilation manager [Blu95].

$$
\begin{array}{lll}
\textit{(ctypes)} & \tau & ::= \mathsf{int} \mid t \mid \tau_1 \rightarrow \tau_2 \\
\textit{(mtypes)} & \sigma & ::= \mathsf{V}(\tau) \mid \sigma_1 \times \cdots \times \sigma_n \mid \sigma_1 \rightarrow \sigma_2 \mid \forall t.\sigma \\
\textit{(inline)} & a & ::= \quad \mid \mathtt{i} \\
\textit{(cterms)} & c & ::= n \mid z \mid \pi_{\mathsf{v}} x \mid \lambda^a z{:}\tau.c \mid c_1 c_2 \\
\textit{(mterms)} & m & ::= x \mid \iota_{\mathsf{v}}(c) \mid \langle x_1,\ldots,x_n \rangle \mid \pi_i x \mid \lambda x{:}\sigma.m \mid @x_1 x_2 \mid \Lambda t.m \mid x[\tau] \\
 & & \quad \mid \mathsf{let}\ x = m_1\ \mathsf{in}\ m_2
\end{array}
$$

Figure 3.2: Syntax for the source calculus SRC

3.2 The source calculus SRC

This section formally defines our source language SRC which is a variant of the polymorphic lambda calculus System-F [Gir72, Rey74]. SRC differs from System-F in that it has inlining annotations on the core functions and it has a stratified structure with a polymorphic module language layered on top of a monomorphic core language. Also the module language uses A-normal form [FSDF93] which means that all intermediate values need to be named via let-binding, thus making all sharing between expressions explicit.

The syntax of SRC is given in Fig. 3.2. Here, a SRC program is just a module term (m). Each module term can be either a variable (x), a structure ($\iota_{\mathsf{v}}(c)$) consisting of a single core term (c), a compound module consisting of a collection of other modules ($\langle x_1,\ldots,x_n \rangle$), an i-th component from another module ($\pi_i x$), a parameterized module (over other modules: $\lambda x{:}\sigma.m$ or over types: $\Lambda t.m$), a module application (over other modules: $@x_1 x_2$ or over types: $x[\tau]$), or a let declaration.

Because the module language can already express polymorphic functions, we intentionally restrict the core language to be a simply typed lambda calculus. A core term can be either an integer constant (n), a variable (z), a value field of a module ($\pi_{\mathsf{v}} x$), a function definition ($\lambda^a z{:}\tau.c$) with inlining annotation (a), or a function application ($c_1 c_2$).

A module type can either be a singleton-value type ($\mathsf{V}(\tau)$ which refers to a module consisting of a core term of type τ), a compound module type ($\sigma_1 \times \cdots \times \sigma_n$ with n sub-modules, each with type σ_i for $i = 1,\ldots,n$), or a parameterized module (over other modules: $\sigma_1 \rightarrow \sigma_2$ or over types: $\forall t.\sigma$). A core type can be either the integer type (int), a type

$$\begin{array}{lccl} \text{kind environment} & \Delta & ::= & \cdot \mid \Delta, t \\ \text{type environment} & \Gamma & ::= & \cdot \mid \Gamma z, \tau{:} \mid \Gamma x, \sigma{:} \end{array}$$

$\boxed{\Delta \vdash \tau \;\text{ and }\; \Delta \vdash \sigma \;\text{ and }\; \Delta \vdash \Gamma}$

$$\frac{}{\Delta \vdash \mathsf{int}} \qquad \frac{t \in \Delta}{\Delta \vdash t} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2} \qquad \frac{\Delta \vdash \tau}{\Delta \vdash \mathsf{V}(\tau)} \qquad \frac{\Delta \vdash \sigma_i \quad (1 \le i \le n)}{\Delta \vdash \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\Delta \vdash \sigma_1 \quad \Delta \vdash \sigma_2}{\Delta \vdash \sigma_1 \to \sigma_2} \qquad \frac{\Delta, t \vdash \sigma}{\Delta \vdash \forall t.\sigma} \qquad \frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \tau}{\Delta \vdash \Gamma z, \tau{:}} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \sigma}{\Delta \vdash \Gamma x, \sigma{:}}$$

$\boxed{\Delta; \Gamma \vdash m : \sigma \;\text{ and }\; \Delta; \Gamma \vdash c : \tau}$

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash n : \mathsf{int}} \qquad \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash z : \Gamma(z)} \qquad \frac{\Delta; \Gamma \vdash x : \mathsf{V}(\tau)}{\Delta; \Gamma \vdash \pi_{\mathsf{v}} x : \tau}$$

$$\frac{\Delta; \Gamma z, \tau_1 \vdash c : \tau_2}{\Delta; \Gamma \vdash \lambda^a z{:}\tau_1.c : \tau_1 \to \tau_2} \qquad \frac{\Delta; \Gamma \vdash c_1 : \tau_2 \to \tau_1 \quad \Delta; \Gamma \vdash c_2 : \tau_2}{\Delta; \Gamma \vdash c_1 c_2 : \tau_1}$$

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash x : \Gamma(x)} \qquad \frac{\Delta; \Gamma \vdash c : \tau}{\Delta; \Gamma \vdash \iota_{\mathsf{v}}(c) : \mathsf{V}(\tau)} \qquad \frac{\Delta; \Gamma \vdash x_i : \sigma_i \quad (1 \le i \le n)}{\Delta; \Gamma \vdash \langle x_1, \ldots, x_n \rangle : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\Delta; \Gamma \vdash x : \sigma_1 \times \cdots \times \sigma_n \quad 1 \le i \le n}{\Delta; \Gamma \vdash \pi_i x : \sigma_i} \qquad \frac{\Delta; \Gamma x, \sigma_1 \vdash m : \sigma_2}{\Delta; \Gamma \vdash \lambda x{:}\sigma_1.m : \sigma_1 \to \sigma_2}$$

$$\frac{\Delta; \Gamma \vdash x_1 : \sigma_2 \to \sigma_1 \quad \Delta; \Gamma \vdash x_2 : \sigma_2}{\Delta; \Gamma \vdash @x_1\, x_2 : \sigma_1} \qquad \frac{\Delta, t; \Gamma \vdash m : \sigma}{\Delta; \Gamma \vdash \Lambda t.m : \forall t.\sigma}$$

$$\frac{\Delta; \Gamma \vdash x : \forall t.\sigma}{\Delta; \Gamma \vdash x[\tau] : \{\tau/t\}\sigma} \qquad \frac{\Delta; \Gamma \vdash m_1 : \sigma_1 \quad \Delta; \Gamma x, \sigma_1 \vdash m_2 : \sigma_2}{\Delta; \Gamma \vdash \mathsf{let}\ x = m_1\ \mathsf{in}\ m_2 : \sigma_2}$$

Figure 3.3: Static semantics for SRC

variable, or a function type $(\tau_1 \to \tau_2)$. The singleton-value type $\mathsf{V}(\tau)$ is used to distinguish between cases like $\mathsf{V}(\mathsf{int} \to \mathsf{int})$ and $\mathsf{V}(\mathsf{int}) \to \mathsf{V}(\mathsf{int})$.

The SRC language was chosen to be expressive enough to exhibit the main difficulties that an optimizer based on staged computation might encounter. The language is split between the module and the core languages because the inliner needs to use two different compilation strategies. Of course, we could merge the two languages and annotate the terms to indicate whether or not to treat them like module code. Recent work [Sha98, Sha99, HMM90] has shown that Standard ML can be compiled into a SRC-like typed intermediate language.

Figure 3.3 gives the static semantics for SRC. The environment Δ is the list of bound type variables; the type environment Γ maps both core and module variables to their respective types. Both the type- and the term-formation rules are rather straight-forward. The language is predicative in that the module language supports polymorphism but type variables can only be instantiated to core types. SRC uses a call-by-value semantics (omitted since it is straightforward); it is easy to show that the typing system for SRC is sound with respect to the corresponding dynamic semantics.

The most interesting feature of SRC is the inlining annotation a. The annotation `i` means that the underlying λ-expression should be inlined if possible while the empty annotation means it should not. Notice that we do not track inlining annotations in the types; a core function $\lambda^a z{:}\tau.c$ has the same type whether the annotation a is `i` or empty.

This design choice is deliberate. We believe inlining annotations should be made as non-intrusive as possible. Tracking them in the types would significantly complicate the module language; for example, we would have to add binding-time polymorphism in the type if we want to apply a functor to modules with different inlining annotations (but with the same signature otherwise).

When compiling SML to a SRC-like language, the SML module language maps to the SRC module language as expected, but the SML core language does not map only to the SRC core language: polymorphic core SML functions translate to module-level type

$$\begin{array}{llll}
\textit{(kind)} & b, k & ::= & \mathsf{s} \mid \mathsf{d} \\
\textit{(type)} & \sigma & ::= & \mathsf{int} \mid t \mid \sigma_1 \times^b \cdots \times \sigma_n \mid \sigma_1 \xrightarrow{b} \sigma_2 \mid \forall^b t{:}k.\sigma \\
\textit{(term)} & e & ::= & v \mid \pi_i^b v \mid @^b v_1\, v_2 \mid v[\sigma]^b \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
\textit{(value)} & v & ::= & n \mid x \mid \langle v_1, \ldots, v_n \rangle^b \mid \lambda^b x{:}\sigma.e \mid \Lambda^b t{:}k.e
\end{array}$$

Figure 3.4: Syntax for the target calculus TLC

abstractions (together with a core-level function) in SRC. This does not introduce any problem, however; since polymorphic recursion is not available, type instantiations can be done statically, or hoisted to the top-level [SS98].

3.3 The target calculus TLC

This section formally defines our target language TLC. As a typed intermediate language, TLC is is essentially a hybrid of System-F in A-normal form [FSDF93] and the two-level lambda calculus $\lambda 2sd$ by Moggi [Mog97].

The syntax of TLC is given in Fig. 3.4. A TLC term (e) can be either a value (v), a record selection, a function application, a type application, or a let expression. A TLC value (v) is either an integer constant, a variable, an n-tuple, a function, or a type function. A TLC type is either the integer type, a type variable, a record type, a function type, or a polymorphic type. Many of these have a binding-time annotation (called “kind”) that can either be s for static code (evaluated at compile-time) or d for dynamic code (evaluated at run-time).

Compared to SRC, TLC replaces inlining hints on core functions with staging annotations on tuples, functions and type-abstractions and merges the module and the core languages since the distinction between the two is only needed to direct the translation from SRC. Notice also how $\forall$ types have two binding-time annotations, one for the type abstraction itself and another that constrains the possible types it can be instantiated to.

To simplify the presentation, we force the ground types (i.e., int) to be considered as dynamic. This is justified by the fact that we are only interested in function-level reductions.

We would lift this restriction if we wanted to model constant propagation.

In the rest of this paper, we will also use the following syntactic sugar:

$$
\begin{aligned}
\pi_i^b e &\equiv \mathsf{let}\ x = e\ \mathsf{in}\ \pi_i^b x \\
e[\sigma]^b &\equiv \mathsf{let}\ x = e\ \mathsf{in}\ x[\sigma]^b \\
@^b e_1\, e_2 &\equiv \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ @^b x_1\, x_2 \\
\langle e_1, \ldots, e_n\rangle^b &\equiv \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = e_n\ \mathsf{in}\ \langle x_1, \ldots, x_n\rangle^b \\
\lambda^b \langle x_1, \ldots, x_n\rangle^{b'}{:}\sigma.e &\equiv \lambda^b x{:}\sigma.\mathsf{let}\ x_1 = \pi_1^{b'} x\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = \pi_n^{b'} x\ \mathsf{in}\ e
\end{aligned}
$$

Essentially, we will put an e term where only v is allowed, leaving the let transformation implicit and we will use a pattern-matching variant of let; we will also assume that alpha-renaming is used so variables are never shadowed.

Figure 3.5 gives the typing rules for TLC. In addition to the usual type safety, these rules also ensure binding-time correctness. Here the kind environment Δ maps type variables to their binding time; the type environment Γ maps variables to their types. To enforce the usual restriction that no dynamic entity can contain or manipulate a static value, types are classified as being either of dynamic kind or static kind, with subkinding such that a type σ of dynamic kind can also be considered to have static kind but not vice versa.

Figures 3.6 to 3.9 give the dynamic semantics for TLC as a set of primitive reductions and single-step evaluation relations that determine where those reductions can be applied. Figure 3.6 defines the primitive static reduction $e \leadsto_{\mathsf{s}} e'$. Figure 3.7 defines the single-step partial evaluation $e \mapsto_{\mathsf{s}} e'$ together with the corresponding $v \mapsto_{\mathsf{s}}^{\mathsf{v}} v'$ used for values. Note how partial evaluation in this language amounts to reducing all the static redexes of a term. Figures 3.8 and 3.9 show the corresponding reductions $\leadsto$ and $\mapsto$ of a standard call-by-value evaluator. In contrast to the partial evaluation case, those reductions apply to both static and dynamic redexes but only to the outermost ones.

TLC is a variant of Moggi's λ_c [Mog88] restricted to A-normal form; the primitive reduction relations in Fig. 3.6 and 3.8 are same as that for λ_c (except that we added type applications and removed η-reductions). We claim that it is easy to show that the type

$$\begin{array}{llcl} \text{kind environment} & \Delta & ::= & \cdot \mid \Delta, t{:}\kappa \\ \text{type environment} & \Gamma & ::= & \cdot \mid \Gamma x, \sigma : \end{array}$$

$\boxed{b_1 \leq b_2}$

$$\mathsf{d} \leq \mathsf{d} \qquad \mathsf{d} \leq \mathsf{s} \qquad \mathsf{s} \leq \mathsf{s}$$

$\boxed{\Delta \vdash \sigma : \kappa \text{ and } \Delta \vdash \Gamma}$

$$\Delta \vdash \mathsf{int} : \mathsf{d} \qquad \Delta \vdash t : \Delta(t) \qquad \frac{\Delta \vdash \sigma : b_1 \quad b_1 \leq b_2}{\Delta \vdash \sigma : b_2} \qquad \frac{\Delta \vdash \sigma_i : b \quad (1 \leq i \leq n)}{\Delta \vdash \sigma_1 \times^b \cdots \times \sigma_n : b}$$

$$\frac{\Delta \vdash \sigma_1 : b \quad \Delta \vdash \sigma_2 : b}{\Delta \vdash \sigma_1 \xrightarrow{b} \sigma_2 : b} \qquad \frac{\Delta, t{:}k \vdash \sigma : b \quad k \leq b}{\Delta \vdash \forall^b t{:}k.\sigma : b} \qquad \frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \sigma : \mathsf{s}}{\Delta \vdash \Gamma x, \sigma :}$$

$\boxed{\Delta; \Gamma \vdash e : \sigma \text{ and } \Delta; \Gamma \vdash v : \sigma}$

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash n : \mathsf{int}} \qquad \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash x : \Gamma(x)} \qquad \frac{\Delta; \Gamma \vdash v_1 : \sigma_2 \xrightarrow{b} \sigma_1 \quad \Delta; \Gamma \vdash v_2 : \sigma_2}{\Delta; \Gamma \vdash @^b v_1\, v_2 : \sigma_1}$$

$$\frac{\Delta; \Gamma \vdash v_i : \sigma_i \quad (1 \leq i \leq n) \quad \Delta \vdash \sigma_1 \times^b \cdots \times \sigma_n : b}{\Delta; \Gamma \vdash \langle v_1, \ldots, v_n \rangle^b : \sigma_1 \times^b \cdots \times \sigma_n} \qquad \frac{\Delta; \Gamma \vdash v : \sigma_1 \times^b \cdots \times \sigma_n \quad 1 \leq i \leq n}{\Delta; \Gamma \vdash \pi_i^b v : \sigma_i}$$

$$\frac{\Delta; \Gamma x, \sigma_1 \vdash e : \sigma_2 \quad \Delta \vdash \sigma_1 \xrightarrow{b} \sigma_2 : b}{\Delta; \Gamma \vdash \lambda^b x{:}\sigma_1.e : \sigma_1 \xrightarrow{b} \sigma_2} \qquad \frac{\Delta; \Gamma \vdash e_1 : \sigma_1 \quad \Delta; \Gamma x, \sigma_1 \vdash e_2 : \sigma_2}{\Delta; \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \sigma_2}$$

$$\frac{\Delta, t{:}k; \Gamma \vdash e : \sigma \quad \Delta \vdash \forall^b t{:}k.\sigma : b}{\Delta; \Gamma \vdash \Lambda^b t{:}k.e : \forall^b t{:}k.\sigma} \qquad \frac{\Delta; \Gamma \vdash v : \forall^b t{:}k.\sigma_2 \quad \Delta \vdash \sigma_1 : k}{\Delta; \Gamma \vdash v[\sigma_1]^b : \{\sigma_1/t\}\sigma_2}$$

Figure 3.5: Static semantics for TLC

system for TLC is sound and the static reduction $\mapsto_{\mathsf{s}}$ is strongly normalizing and confluent. We can thus define a partial evaluation function $\mathcal{P}_{\mathsf{e}}(e)$ that returns the static normal form of e. Similarly it is easy to show that $\mapsto$ is confluent, so we can also define a partial function $\mathcal{R}_{\mathsf{e}}(e)$ which does the standard evaluation of e. $\mathcal{R}_{\mathsf{e}}(e)$ happens to be a total function, but we do not rely on it.

$$\mathcal{P}_{\mathsf{e}}(e) = e' \text{such that } e \mapsto_{\mathsf{s}}^* e' \text{ and there is no } e'' \text{ for which } e' \mapsto_{\mathsf{s}} e''$$

$$\mathcal{R}_{\mathsf{e}}(e) = e' \text{such that } e \mapsto^* e' \text{ and there is no } e'' \text{ for which } e' \mapsto e''$$

$$\begin{array}{lll}
(\beta_\lambda) & @^{\mathsf{s}}(\lambda^{\mathsf{s}}x{:}\sigma.e)\,(v) \leadsto_{\mathsf{s}} \{v/x\}e & \\
(\beta_\Lambda) & (\Lambda^{\mathsf{s}}t{:}k.e)[\sigma]^{\mathsf{s}} \leadsto_{\mathsf{s}} \{\sigma/t\}e & \\
(\pi) & \pi_i^{\mathsf{s}}\langle v_1,\ldots,v_n\rangle^{\mathsf{s}} \leadsto_{\mathsf{s}} v_i & \text{if } 1 \le i \le n \\
\textit{(let)} & \mathsf{let}\ x = v\ \mathsf{in}\ e \leadsto_{\mathsf{s}} \{v/x\}e & \\
\textit{(asc)} & \mathsf{let}\ x_2 = (\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2)\ \mathsf{in}\ e_3 & \\
 & \quad \leadsto_{\mathsf{s}}\ \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ e_3 &
\end{array}$$

Figure 3.6: Primitive static reduction for TLC

$$\begin{array}{rcc}
e \leadsto_{\mathsf{s}} e' & \Rightarrow & e \mapsto_{\mathsf{s}} e' \\
e \mapsto_{\mathsf{s}} e' & \Rightarrow & \mathsf{let}\ x = e\ \mathsf{in}\ e_2 \mapsto_{\mathsf{s}} \mathsf{let}\ x = e'\ \mathsf{in}\ e_2 \\
e \mapsto_{\mathsf{s}} e' & \Rightarrow & \mathsf{let}\ x = e_2\ \mathsf{in}\ e \mapsto_{\mathsf{s}} \mathsf{let}\ x = e_2\ \mathsf{in}\ e' \\
e \mapsto_{\mathsf{s}} e' & \Rightarrow & \lambda^b x{:}\sigma.e \mapsto_{\mathsf{s}}^{\mathsf{v}} \lambda^b x{:}\sigma.e' \\
e \mapsto_{\mathsf{s}} e' & \Rightarrow & \Lambda^b t{:}k.e \mapsto_{\mathsf{s}}^{\mathsf{v}} \Lambda^b t{:}k.e' \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & \langle v_1,\ldots,v,\ldots,v_n\rangle^b \mapsto_{\mathsf{s}}^{\mathsf{v}} \langle v_1,\ldots,v',\ldots,v_n\rangle^b \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & @^b v\, v_2 \mapsto_{\mathsf{s}} @^b v'\, v_2 \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & @^b v_2\, v \mapsto_{\mathsf{s}} @^b v_2\, v' \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & v[\sigma]^b \mapsto_{\mathsf{s}} v'[\sigma]^b \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & \pi_i^b v \mapsto_{\mathsf{s}} \pi_i^b v'
\end{array}$$

Figure 3.7: Single-step partial evaluation for TLC

$$\begin{array}{llll}
(\leadsto_{\mathsf{s}}) & e & \leadsto e' & \text{if } e \leadsto_{\mathsf{s}} e' \\
(\beta_\lambda) & @^{\mathsf{d}}(\lambda^{\mathsf{d}}x{:}\tau.e)\,(v) & \leadsto \{v/x\}e & \\
(\beta_\Lambda) & (\Lambda^{\mathsf{d}}t{:}k.e)[\tau]^{\mathsf{d}} & \leadsto \{\tau/t\}e & \\
(\pi) & \pi_i^{\mathsf{d}}\langle v_1,\ldots,v_n\rangle^{\mathsf{d}} & \leadsto v_i & \text{if } 1 \le i \le n
\end{array}$$

Figure 3.8: Primitive reduction relation for TLC

$$\begin{array}{rcc}
e \leadsto e' & \Rightarrow & e \mapsto e' \\
e \mapsto e' & \Rightarrow & \mathsf{let}\ x = e\ \mathsf{in}\ e_2 \mapsto \mathsf{let}\ x = e'\ \mathsf{in}\ e_2
\end{array}$$

Figure 3.9: Single-step call-by-value standard evaluation for TLC

where $\mapsto^*$ and $\mapsto_{\mathsf{s}}^*$ are the reflexive transitive closures of $\mapsto$ and $\mapsto_{\mathsf{s}}$. TLC satisfies the following important residualization property:

Theorem 3.3.1 (Residualization)
If $\Delta; \Gamma \vdash e : \sigma$ and $\Delta \vdash \sigma : \mathsf{d}$ and $\forall x \in \mathsf{fv}(e).\ \Delta \vdash \Gamma(x) : \mathsf{d}$, then $\mathcal{P}_{\mathsf{e}}(e)$ is free of any static subterms.

In other words, given an expression e with dynamic type σ, partially evaluating e will inline all of its inlinable functions and result in an expression free of static subterms.

Next we show why inlining does not affect the semantics of the program. We first introduce a notion of semantic equivalence on well-typed TLC values:

Definition 3.3.2 (Equivalence)
If $\cdot; \cdot \vdash v : \sigma$ and $\cdot; \cdot \vdash v' : \sigma$, we say that $v \simeq v'$ if one of the following holds:

(int) $v \equiv v'$.

($\times$) $v = \langle v_1,..., v_n\rangle^b$ *and* $v' = \langle v'_1,..., v'_n\rangle^b$ *and* $\forall i \in [1..n].v_i \simeq v'_i$.

($\rightarrow$) $\sigma = \sigma_1 \xrightarrow{b} \sigma_2$ *and for any value* w *of type* σ_1 *then* $\mathcal{R}_{\mathsf{e}}(@^b v\, w) \simeq \mathcal{R}_{\mathsf{e}}(@^b v'\, w)$.

($\forall$) $\sigma = \forall^b t{:}k.\sigma_1$ *and for any well-formed type* $\sigma_2{:}k$ *then* $\mathcal{R}_{\mathsf{e}}(v[\sigma_2]^b) \simeq \mathcal{R}_{\mathsf{e}}(v'[\sigma_2]^b)$.

The correctness theorem can then be proved by induction over the reduction steps of $\mathcal{P}_{\mathsf{e}}(e)$.

Theorem 3.3.3 (Correctness)
If $\cdot; \cdot \vdash e : \sigma$ and $\cdot \vdash \sigma : \mathsf{d}$ then $\mathcal{R}_{\mathsf{e}}(\mathcal{P}_{\mathsf{e}}(e)) \simeq \mathcal{R}_{\mathsf{e}}(e)$.

3.4 Translation from SRC to TLC

The translation from SRC to TLC involves both staging and splitting, executed in an interleaved manner. Staging translates inlining annotations in the core language into binding-time annotations. It also calls the splitting algorithm to divide each module term into a static part and a dynamic part. The static part is used to propagate inlining information and implement cross-module inlining. In the rest of this section, we first give a quick overview of our approach; we then show how to stage core terms and split module terms; finally, we give the main translation algorithm that links all the parts together.

3.4.1 A quick overview

The translation from SRC to TLC mostly consists of adding staging annotations. This is usually known as binding-time analysis and has been extensively studied in the partial evaluation community.

One desirable goal is to make sure that binding-time annotations do not hide opportunities for static evaluation. For example, let's take the inlinable compose function o defined as follows:

$$o = \lambda^{\mathsf{i}} f{:}\tau_1 \to \tau_2.\lambda^{\mathsf{i}} g{:}\tau_2 \to \tau_3.\lambda x{:}\tau_1.g(f x)$$

When translating it, we probably do not want to assign it the following type:

$$o : (\tau_1 \xrightarrow{\mathsf{d}} \tau_2) \xrightarrow{\mathsf{s}} (\tau_2 \xrightarrow{\mathsf{d}} \tau_3) \xrightarrow{\mathsf{s}} (\tau_1 \xrightarrow{\mathsf{d}} \tau_3)$$

(i.e. a static function that composes two dynamic functions) since it would force us to make sure that all the functions passed to it are not inlinable, which mostly defeats the purpose of inlining it in the first place. Now clearly, if we mark it as:

$$o : (\tau_1 \xrightarrow{\mathsf{s}} \tau_2) \xrightarrow{\mathsf{s}} (\tau_2 \xrightarrow{\mathsf{s}} \tau_3) \xrightarrow{\mathsf{s}} (\tau_1 \xrightarrow{\mathsf{d}} \tau_3)$$

that will make it impossible to call it with a non-inlinable function. We could work around this problem by using polymorphism at the binding-time level [HM94, GSSS01], but we decided to keep our calculus simple. With monomorphic staging annotations, we have two options: code duplication to provide a poor man's polymorphic binding-time, or coercions in the form of binding-time improvements [DMP96, Dan96].

A compiler needs to be very careful about duplicating code so we decided to use coercions instead, especially since they provide us with a lot of flexibility. More specifically, we can completely avoid the need for a full-blown binding-time analysis and use a simple one-pass translation instead, by optimistically marking s any place that might need to accommodate a static value and inserting coercions when needed, just like the unboxing coercions [Ler92, Sha97a]. It also allows us to simplify our types: all types are either (completely) dynamic or completely static. In effect, we are using polymorphism but encoded using Leroy's coercions.

3.4.2 Staging the core

Staging could be done via any kind of binding-time analysis [Con93, BW95], but this would be too costly for our application, so instead of performing global code analysis to add the annotations, we add them in a single traversal of the code using only local information. In order to maximize the amount of static computation, we make extensive use of binding-time improvements [DMP96].

Binding-time improvements are usually some form of η-redexes that coerce an object between its static and dynamic representations. They improve the binding-time annotations by allowing values to be used statically at one place and dynamically at another and even to make this choice "dynamically" during specialization.

Staging is then simple: based on the inlining annotations, SRC terms can either be translated to completely static or dynamic entities (except for int, which is always dynamic). Because inlining annotations are not typechecked in SRC, the resulting TLC terms may use dynamic subterms in a static context or vice versa. We insert coercions whenever there is such a mismatch.

We define two type-translation functions $|\cdot|^{\mathsf{s}}$ and $|\cdot|^{\mathsf{d}}$ that turn any SRC type into either its fully static or its fully dynamic TLC equivalent, and two coercion functions $\downarrow^{\tau} : |\tau|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\tau|^{\mathsf{d}}$ and $\uparrow^{\tau} : |\tau|^{\mathsf{d}} \xrightarrow{\mathsf{s}} |\tau|^{\mathsf{s}}$. Those coercions (and corresponding type translations) could simply be:

$$\begin{array}{lll} \downarrow^{\mathsf{int}} x & = x & |\mathsf{int}|^{\mathsf{d}} = \mathsf{int} \\ \downarrow^{\tau_1 \to \tau_2} x & = \lambda^{\mathsf{d}} x_1{:}|\tau_1|^{\mathsf{d}}.\ \downarrow^{\tau_2}(@^{\mathsf{s}} x\,(\uparrow^{\tau_1} x_1)) & |\tau_1 \to \tau_2|^{\mathsf{d}} = |\tau_1|^{\mathsf{d}} \xrightarrow{\mathsf{d}} |\tau_2|^{\mathsf{d}} \\ & \ldots & \ldots \\ \uparrow^{\mathsf{int}} x & = x & |\mathsf{int}|^{\mathsf{s}} = \mathsf{int} \\ \uparrow^{\tau_1 \to \tau_2} x & = \lambda^{\mathsf{s}} x_1{:}|\tau_1|^{\mathsf{s}}.\ \uparrow^{\tau_2}(@^{\mathsf{d}} x\,(\downarrow^{\tau_1} x_1)) & |\tau_1 \to \tau_2|^{\mathsf{s}} = |\tau_1|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\tau_2|^{\mathsf{s}} \\ & \ldots & \ldots \end{array}$$

But this would run the risk of introducing unexpected code duplication.

Spurious copies A naive coercion of a static function to its dynamic equivalent tends to introduce static redexes which cause the function to be inlined unnecessarily at the place where it escapes. Consider the following piece of SRC code:

$$\begin{array}{l} \mathsf{let}\ id\ = \lambda^{\mathsf{i}} x{:}\mathsf{int}.x \\ \quad big = \lambda f{:}\mathsf{int} \to \mathsf{int}.\ ...big\ body... \\ \mathsf{in}\ \langle id, @\, big\, id \rangle \end{array}$$

A simple-minded staging scheme would turn it into:

$$\begin{array}{l} \mathsf{let}\ id\ = \lambda^{\mathsf{s}} x{:}\mathsf{int}.x \\ \quad big = \lambda^{\mathsf{d}} f{:}\mathsf{int} \xrightarrow{\mathsf{d}} \mathsf{int}.\ ...big\ body... \\ \mathsf{in}\ \langle\ \downarrow^{\mathsf{int}\to\mathsf{int}} id, @^{\mathsf{d}} big\,(\downarrow^{\mathsf{int}\to\mathsf{int}} id) \rangle^{\mathsf{d}} \end{array}$$

where the coercions get expanded to:

$$\begin{array}{l} \mathsf{let}\ id\ = \lambda^{\mathsf{s}} x{:}\mathsf{int}.x \\ \quad big = \lambda^{\mathsf{d}} f{:}\mathsf{int} \xrightarrow{\mathsf{d}} \mathsf{int}.\ ...big\ body... \\ \mathsf{in}\ \langle \lambda^{\mathsf{d}} x{:}\mathsf{int}.@^{\mathsf{s}} id\, x, @^{\mathsf{d}} big\,(\lambda^{\mathsf{d}} x{:}\mathsf{int}.@^{\mathsf{s}} id\, x) \rangle^{\mathsf{d}} \end{array}$$

Note that the two escaping uses of *id* have been turned now into η-redexes where *id* is called directly. Thus specialization will happily inline two copies of *id* even though

$$
\begin{aligned}
|\mathsf{int}|^{\mathsf{d}} &= \mathsf{int} & |\mathsf{int}|^{\mathsf{s}} &= \mathsf{int} \\
|\tau_1 \to \tau_2|^{\mathsf{d}} &= |\tau_1|^{\mathsf{d}} \xrightarrow{\mathsf{d}} |\tau_2|^{\mathsf{d}} & |\tau_1 \to \tau_2|^{\mathsf{s}} &= |\tau_1|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\tau_2|^{\mathsf{s}} \times^{\mathsf{s}} |\tau_1 \to \tau_2|^{\mathsf{d}}
\end{aligned}
$$

$$
\begin{aligned}
\downarrow^{\tau} &: |\tau|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\tau|^{\mathsf{d}} & \uparrow^{\tau} &: |\tau|^{\mathsf{d}} \xrightarrow{\mathsf{s}} |\tau|^{\mathsf{s}} \\
\downarrow^{\mathsf{int}} x &= x & \uparrow^{\mathsf{int}} x &= x \\
\downarrow^{\tau_1 \to \tau_2} x &= \pi_2^{\mathsf{s}} x & \uparrow^{\tau_1 \to \tau_2} x &= \langle \lambda^{\mathsf{s}} x_1{:}|\tau_1|^{\mathsf{s}}.\ \uparrow^{\tau_2}(@^{\mathsf{d}} x\,(\downarrow^{\tau_1} x_1)), x \rangle^{\mathsf{s}}
\end{aligned}
$$

$$
\begin{aligned}
|\,\Delta; \Gamma; \Sigma\,| &= \Delta; \Gamma' \\
\text{where } \Gamma' &= \{x{:}|\Gamma(x)|^{\mathsf{s}} \mid x \in \mathsf{dom}(\Gamma)\} \cup \{z{:}|\tau|^{b} \mid \tau = \Gamma(z) \text{ and } b = \Sigma(z)\} \\
&\text{and } (\textit{binding-time environment})\ \Sigma ::= \cdot \mid \Sigma, z{:}b
\end{aligned}
$$

Figure 3.10: Core type translations $|\cdot|$ and coercions $\downarrow$ and $\uparrow$.

no optimization will be enabled since both uses are really escaping. We do not want to introduce such wasteful code duplication.

In other words, we want to ensure that there can be only one non-inlined copy of any function, shared among all its escaping uses. To this end, we must arrange for $\downarrow^{\sigma}$ not to introduce spurious redexes. We could simply introduce a special coerce primitive operation with an ad-hoc treatment in the partial-evaluator, but depending on the semantics chosen (e.g. binding-time erasure) it can end up leaking static code to run-time, introducing unwanted run-time penalties and it does not easily solve the problem of ensuring a unique dynamic copy of a function, even in the presence of cross-module inlining.

So we decided to choose a more sophisticated representation for the static translation of a function, where each static function is now represented as a pair of the real static function and the already-coerced dynamic function. Now $\downarrow^{\sigma_1 \to \sigma_2}$ becomes π_2^{s} and the only real coercion happens once, making it clear that only one instance of the dynamic version will exist. The definition of our type translations $|\cdot|^{\mathsf{s}}$ and $|\cdot|^{\mathsf{d}}$ and coercions $\downarrow^{\tau}$ and $\uparrow^{\tau}$ for the core calculus is shown in Fig. 3.10. The previous example is now staged as follows:

$$
\begin{aligned}
&\mathsf{let}\ id = \langle \lambda^{\mathsf{s}} x{:}\mathsf{int}.x, \lambda^{\mathsf{d}} x{:}\mathsf{int}.x \rangle^{\mathsf{s}} \\
&\qquad big = \lambda^{\mathsf{d}} f{:}\mathsf{int} \xrightarrow{\mathsf{d}} \mathsf{int}.\ ...big\ body... \\
&\mathsf{in}\ \langle \downarrow^{\mathsf{int}\to\mathsf{int}} id, @^{\mathsf{d}} big\,(\downarrow^{\mathsf{int}\to\mathsf{int}} id) \rangle^{\mathsf{d}}
\end{aligned}
$$

$\boxed{\Delta;\Gamma;\Sigma \vdash [\![c:\tau]\!] \stackrel{b}{\Rightarrow} e}$ such that $|\Delta;\Gamma;\Sigma| \vdash e : |\tau|^b$

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma;\Sigma \vdash [\![n:\mathsf{int}]\!] \stackrel{b}{\Rightarrow} n} \qquad \frac{\Delta \vdash \Gamma \quad \Sigma(z) = b}{\Delta;\Gamma;\Sigma \vdash [\![z:\Gamma(\tau)]\!] \stackrel{b}{\Rightarrow} z} \qquad \frac{\Delta;\Gamma \vdash x : \mathsf{V}(\tau)}{\Delta;\Gamma;\Sigma \vdash [\![\pi_{\mathsf{v}} x:\tau]\!] \stackrel{\mathsf{s}}{\Rightarrow} x}$$

$$\frac{\Delta;\Gamma z,\tau_1;\Sigma z,\mathsf{s} \vdash [\![c:\tau_2]\!] \stackrel{\mathsf{s}}{\Rightarrow} e}{\begin{array}{l}\Delta;\Gamma;\Sigma \vdash [\![\lambda^{\mathsf{i}} z{:}\tau_1.c:\tau_1 \to \tau_2]\!] \stackrel{\mathsf{s}}{\Rightarrow} \\ \mathsf{let}\ x_{\mathsf{s}} = \lambda^{\mathsf{s}} z{:}|\tau_1|^{\mathsf{s}}.e \\ \quad\ x_{\mathsf{d}} = \lambda^{\mathsf{d}} z{:}|\tau_1|^{\mathsf{d}}.\downarrow^{\tau_2}(@^{\mathsf{s}} x_{\mathsf{s}}\,(\uparrow^{\tau_1} z)) \\ \mathsf{in}\ \langle x_{\mathsf{s}}, x_{\mathsf{d}}\rangle^{\mathsf{s}}\end{array}} \qquad \frac{\Delta;\Gamma z,\tau_1;\Sigma z,\mathsf{d} \vdash [\![c:\tau_2]\!] \stackrel{\mathsf{d}}{\Rightarrow} e}{\Delta;\Gamma;\Sigma \vdash [\![\lambda z{:}\tau_1.c:\tau_1 \to \tau_2]\!] \stackrel{\mathsf{d}}{\Rightarrow} \lambda^{\mathsf{d}} z{:}|\tau_1|^{\mathsf{d}}.e}$$

$$\frac{\Delta;\Gamma;\Sigma \vdash [\![c_1:\tau_1 \to \tau_2]\!] \stackrel{\mathsf{d}}{\Rightarrow} e_1 \quad \Delta;\Gamma;\Sigma \vdash [\![c_2:\tau_1]\!] \stackrel{\mathsf{d}}{\Rightarrow} e_2}{\Delta;\Gamma;\Sigma \vdash [\![c_1 c_2:\tau_2]\!] \stackrel{\mathsf{d}}{\Rightarrow} @^{\mathsf{d}} e_1\, e_2} \qquad \frac{\Delta;\Gamma;\Sigma \vdash [\![c:\tau]\!] \stackrel{\mathsf{d}}{\Rightarrow} e}{\Delta;\Gamma;\Sigma \vdash [\![c:\tau]\!] \stackrel{\mathsf{s}}{\Rightarrow} \uparrow^{\tau} e}$$

$$\frac{\Delta;\Gamma;\Sigma \vdash [\![c_1:\tau_1 \to \tau_2]\!] \stackrel{\mathsf{s}}{\Rightarrow} e_1 \quad \Delta;\Gamma;\Sigma \vdash [\![c_2:\tau_1]\!] \stackrel{\mathsf{s}}{\Rightarrow} e_2}{\Delta;\Gamma;\Sigma \vdash [\![c_1 c_2:\tau_2]\!] \stackrel{\mathsf{s}}{\Rightarrow} @^{\mathsf{s}}(\pi_1^{\mathsf{s}} e_1)\, e_2} \qquad \frac{\Delta;\Gamma;\Sigma \vdash [\![c:\tau]\!] \stackrel{\mathsf{s}}{\Rightarrow} e}{\Delta;\Gamma;\Sigma \vdash [\![c:\tau]\!] \stackrel{\mathsf{d}}{\Rightarrow} \downarrow^{\tau} e}$$

Figure 3.11: Core code translation.

Since the coercion $\downarrow^{\mathsf{int}\to\mathsf{int}}$ is now π_2^{s}, it just selects the dynamic version of *id*, with no code duplication.

Partial evaluators have long used such paired representation in their specializer for similar reasons [Asa99], although our case is slightly different in that the pairs are explicit in the program being specialized rather than used internally by the specializer.

This pairing approach can also be seen as a poor man's polymorphic binding-time where we only allow the two extreme cases (all dynamic or all static). Minamide and Garrigue [MG98] used the same pairing approach when attempting to avoid the problem of accumulative coercion wrappers that appears when unboxing coercions are used to reconcile polymorphism and specialized data representation.

The staging algorithm is shown in Fig. 3.11. The judgment $\Delta;\Gamma;\Sigma \vdash [\![c:\tau]\!] \stackrel{b}{\Rightarrow} e$ says that a core SRC term c of type τ (under contexts Δ and Γ) is translated into a TLC term e. The environment Σ maps core variables to the binding-time of the corresponding variable in e. The b on the arrow indicates whether a static or a dynamic term e is expected.

Most rules come in two forms, depending on whether the context expects a dynamic or

$\boxed{\Delta;\Gamma \vdash^{\mathsf{split}} [\![e]\!] \xRightarrow{\sigma} E_{\mathsf{e}}\,;\,e_{\mathsf{i}}}$ where $\Delta;\Gamma \vdash e : |\sigma|^{\mathsf{s}}$

$$\frac{\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ e_3]\!] \xRightarrow{\sigma} E_{\mathsf{e}}\,;\,e_{\mathsf{i}}}{\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x_2 = \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2\ \mathsf{in}\ e_3]\!] \xRightarrow{\sigma} E_{\mathsf{e}}\,;\,e_{\mathsf{i}}}\ \text{(sp-asc)}$$

$$\frac{\begin{array}{c} x \notin \mathsf{fv}(e_{\mathsf{i}}) \vee \Delta \vdash \sigma_1 : \mathsf{d} \\ \Delta;\Gamma \vdash e_1 : \sigma_1 \quad \Delta;\Gamma x, \sigma_1 \vdash^{\mathsf{split}} [\![e_2]\!] \xRightarrow{\sigma} E_{\mathsf{e}}\,;\,e_{\mathsf{i}} \end{array}}{\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] \xRightarrow{\sigma} (\mathsf{let}\ x = e_1\ \mathsf{in}\ E_{\mathsf{e}})\,;\,e_{\mathsf{i}}}\ \text{(sp-share)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \sigma_1 \quad \Delta;\Gamma x, \sigma_1 \vdash^{\mathsf{split}} [\![e_2]\!] \xRightarrow{\sigma} E_{\mathsf{e}}\,;\,e_{\mathsf{i}}}{\begin{array}{c} \Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] \xRightarrow{\sigma} \\ (\mathsf{let}\ x = e_1\ \mathsf{in}\ E_{\mathsf{e}})\ ;\ (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_{\mathsf{i}}) \end{array}}\ \text{(sp-dup)}$$

$$\frac{E_{\mathsf{e}} = (\mathsf{let}\ x_{\mathsf{fv}} = \bullet\ \mathsf{in}\ \langle \downarrow^{\sigma} x,\ x_{\mathsf{fv}} \rangle^{\mathsf{d}})}{\Delta;\Gamma \vdash^{\mathsf{split}} [\![x]\!] \xRightarrow{\sigma} E_{\mathsf{e}}\,;\,x}\ \text{(sp-var)} \qquad \frac{\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x = e\ \mathsf{in}\ x]\!] \xRightarrow{\sigma} E_{\mathsf{e}}\,;\,e_{\mathsf{i}}}{\Delta;\Gamma \vdash^{\mathsf{split}} [\![e]\!] \xRightarrow{\sigma} E_{\mathsf{e}}\,;\,e_{\mathsf{i}}}\ \text{(sp-exp)}$$

Figure 3.12: The λ-split algorithm.

static term. The dynamic case is trivial (it corresponds to the no-inlining case so we do not need to do anything) while the static case needs to build the static/dynamic pair (in the λ case) or to extract the static half of the pair before applying it (in the @ case).

3.4.3 Splitting

Module-level functions are typically used differently from core-level functions. They also do not have any inlining annotations thus deserve special treatment during the translation. As noted earlier, it is desirable to mark all the module code as static to "compile it away" or at least, to allow inlining information to flow freely through module boundaries. But that would imply that every single module-level function application gets its own copy of the body, which leads to unnecessary code duplication.

To overcome this difficulty, we use a form of partial inlining inspired from Blume and Appel's λ-splitting [BA97] that splits each function into a static and a dynamic part. It rewrites a TLC expression e into a list of let bindings and copies every inlinable (i.e. static) binding from e into e_{i}, and puts the rest into the expression e_{e} (the e subscript stands for "expansive") in such a way that the two can be combined to get back an expression equivalent to e with $e \simeq \mathsf{let}\ \langle fv \rangle = e_{\mathsf{e}}\ \mathsf{in}\ e_{\mathsf{i}}$ where fv is the list of free variables of e_{i}. Since e_{i} is small, it can be copied wherever e was originally used, while the main part of the code is kept separate in e_{e}.

Basically, e_{i} is just like e but where all the non-inlinable code has been taken out (and the variables that refer to it are thus free), whereas e_{e} is a complete copy of e except that it returns all those values that have been taken out of e_{i}, so it can be used to close over the free variables of e_{i}. Take for example the following expression e where *lookup* is inlinable but *balance* is not (note that the algorithm assumes that the return value of e is completely static):

$$
\begin{aligned}
e = \mathsf{let}\ & \mathit{balance} = \lambda^{\mathsf{d}}\langle t, x\rangle^{\mathsf{d}}{:}\mathit{tree}^{\mathsf{d}} \times^{\mathsf{d}} \mathit{elem}^{\mathsf{d}}.\ldots \\
& \quad \mathit{lookup}\ = \lambda^{\mathsf{s}}\langle t, p\rangle^{\mathsf{s}}{:}\mathit{tree}^{\mathsf{s}} \times^{\mathsf{s}} \mathit{elem}^{\mathsf{s}} \xrightarrow{\mathsf{s}} \mathit{bool}. \\
& \qquad\qquad \mathsf{let}\ x = @^{\mathsf{s}}(@^{\mathsf{s}}\mathit{find}\ p)\,t\ \mathsf{in}\ @^{\mathsf{d}}\mathit{balance}\,\langle\, {\downarrow^{\mathit{tree}}} t, {\downarrow^{\mathit{elem}}} x\rangle^{\mathsf{d}} \\
\mathsf{in}\ & \mathit{lookup}
\end{aligned}
$$

This expression e will be split into a dynamic e_{e} and a static e_{i} where *balance* has been taken out since it is not inlinable. e_{i} will look like:

$$
\begin{aligned}
e_{\mathsf{i}} = \mathsf{let}\ & \mathit{lookup} = \lambda^{\mathsf{s}}\langle t, p\rangle^{\mathsf{s}}{:}\mathit{tree}^{\mathsf{s}} \times^{\mathsf{s}} \mathit{elem}^{\mathsf{s}} \xrightarrow{\mathsf{s}} \mathit{bool}. \\
& \qquad\qquad \mathsf{let}\ x = @^{\mathsf{s}}(@^{\mathsf{s}}\mathit{find}\ p)\,t\ \mathsf{in}\ @^{\mathsf{d}}\mathit{balance}\,\langle\, {\downarrow^{\mathit{tree}}} t, {\downarrow^{\mathit{elem}}} x\rangle^{\mathsf{d}} \\
\mathsf{in}\ & \mathit{lookup}
\end{aligned}
$$

The free variables of e_{i} are provided by e_{e} which carries all the old code and returns all the missing bindings for e_{i} to use. In this example, *balance* is the only free variable, so the result looks like:

$$
\begin{aligned}
e_{\mathsf{e}} = \mathsf{let}\ & \mathit{balance} = \lambda^{\mathsf{d}}\langle t, x\rangle^{\mathsf{d}}{:}\mathit{tree}^{\mathsf{d}} \times^{\mathsf{d}} \mathit{elem}^{\mathsf{d}}.\ldots \\
& \quad \mathit{lookup}\ = \lambda^{\mathsf{s}}\langle t, p\rangle^{\mathsf{s}}{:}\mathit{tree}^{\mathsf{s}} \times^{\mathsf{s}} \mathit{elem}^{\mathsf{s}} \xrightarrow{\mathsf{s}} \mathit{bool}. \\
& \qquad\qquad \mathsf{let}\ x = @^{\mathsf{s}}(@^{\mathsf{s}}\mathit{find}\ p)\,t\ \mathsf{in}\ @^{\mathsf{d}}\mathit{balance}\,\langle\, {\downarrow^{\mathit{tree}}} t, {\downarrow^{\mathit{elem}}} x\rangle^{\mathsf{d}} \\
\mathsf{in}\ & \langle \mathit{balance} \rangle^{\mathsf{d}}
\end{aligned}
$$

And we can combine e_{e} and e_{i} back together with $e \simeq \mathsf{let}\ \langle fv\rangle^{\mathsf{d}} = e_{\mathsf{e}}\ \mathsf{in}\ e_{\mathsf{i}}$ where fv is the list of free variables of e_{i}. We could of course remove *lookup* from e_{e}, but we might need it for something else (as will be shown in the next section) and it is easier to take care of it in a separate dead code elimination pass.

Because e_{e} has to return all the free variables of e_{i}, which are not known until e_{i} is complete, we cannot conveniently build e_{e} directly as we build e_{i}. Instead we build an expression with a hole E_{e} such that $e_{\mathsf{e}} \equiv E_{\mathsf{e}}[\langle fv\rangle^{\mathsf{d}}]$:

$$E_{\mathsf{e}} = \mathsf{let}\ balance = \ \ldots \quad lookup = \ \ldots\ \mathsf{in}\ \langle\bullet\rangle^{\mathsf{d}}$$

Here a TLC term with a hole E is formally defined as follows:

$$E \ ::= \ \mathsf{let}\ x = \bullet\ \mathsf{in}\ e \ \mid \ \mathsf{let}\ x = e\ \mathsf{in}\ E$$

$E[e]$ then fills the hole in E by textually substituting e for $\bullet$ *without* avoiding name capture:

$$(\mathsf{let}\ x = \bullet\ \mathsf{in}\ e)[e_1] \equiv (\mathsf{let}\ x = e_1\ \mathsf{in}\ e)$$
$$(\mathsf{let}\ x = e\ \mathsf{in}\ E)[e_1] \equiv (\mathsf{let}\ x = e\ \mathsf{in}\ E[e_1])$$

The splitting rules are shown in Fig. 3.12. The judgment $\Delta;\Gamma \vdash^{\mathsf{split}} [\![e]\!] \overset{\sigma}{\Longrightarrow} E_{\mathsf{e}}\,;e_{\mathsf{i}}$ states that E_{e} and e_{i} are a valid split of e in contexts Δ and Γ assuming that $e : |\sigma|^{\mathsf{s}}$. The rules only guarantee correctness of the split, but do not specify a deterministic algorithm. In practice, whenever several rules can apply, the splitting algorithm chooses the first rule shown that applies: *sp-asc* is preferred over *sp-share* which is preferred over *sp-dup*, while *sp-exp* is only applied when there is no other choice (i.e. when e is neither a let binding nor a mere variable). This ensures that we return the smallest e_{i}.

The way the rules work is as follows: *sp-asc* together with *sp-exp* turn e into a list of let bindings that ends by returning a variable; *sp-share* copies dynamic bindings to E_{e} but omits them from e_{i} while *sp-dup* copies static bindings to both E_{e} and e_{i}; finally *sp-var* replaces the terminating variable with a hole in E_{e}.

Splitting functions

When splitting a function f, we could apply the above algorithm to the body, and then combine the two results into two functions f_{e} and f_{i}:

$$f = \lambda^{\mathsf{s}} x{:}|\sigma|^{\mathsf{s}}.e \implies \begin{aligned} f_{\mathsf{e}} &= \lambda^{\mathsf{d}} x_{\mathsf{d}}{:}|\sigma|^{\mathsf{d}}.\mathsf{let}\ x = \uparrow^{\sigma} x_{\mathsf{d}}\ \mathsf{in}\ e_{\mathsf{e}} \\ f_{\mathsf{i}} &= \lambda^{\mathsf{s}} x{:}|\sigma|^{\mathsf{s}}.\mathsf{let}\ \langle fv \rangle^{\mathsf{d}} = @^{\mathsf{d}} f_{\mathsf{e}}\,(\downarrow^{\sigma} x)\ \mathsf{in}\ e_{\mathsf{i}} \end{aligned}$$

From then on f_{i} can be used in place of f (assuming that f_{e} is in scope), so that e_{i} will be inlined without having to ever duplicate e_{e}.

As we have seen before when staging inlinable core functions, the static representation of a function is a pair of the dynamic and the static version of that function $f = \langle f_{\mathsf{d}}, f_{\mathsf{s}} \rangle^{\mathsf{s}}$. A similar representation needs to be used for module-level functions. One would be tempted to just use f_{i} for f_{s} and f_{e} for f_{d}, but a bit more work is required. First, we cannot use f_{e} directly because it only returns the free variables of f_{i} instead of the expected return value of f_{d}, but we can simply coerce f_{i} (which has f_{e} as a free variable) to a dynamic value:

$$f_{\mathsf{s}} = f_{\mathsf{i}} \qquad\qquad f_{\mathsf{d}} = \lambda^{\mathsf{d}} x_{\mathsf{d}}{:}|\sigma_1|^{\mathsf{d}}.\downarrow^{\sigma_2}(@^{\mathsf{s}} f_{\mathsf{i}}\,(\uparrow^{\sigma_1} x_{\mathsf{d}}))$$

The problem with this approach is that there might be some code duplication between e_{i} and e_{e}, so f_{d} might contain unnecessary copies of code already existing in f_{e}. To work around this, we slightly change the way splitting is done, so that e_{e} returns not only the free variables of e_{i} but also the original output of e (see the *sp-var* rule):

$$E_{\mathsf{e}} = \mathsf{let}\ \mathit{balance} = \ \ldots \quad \mathit{lookup} = \ \ldots\ \mathsf{in}\ \langle \downarrow^{\sigma} \mathit{lookup}, \bullet \rangle^{\mathsf{d}}$$

Of course, we need to adjust f_{i} so as to select the second component of f_{e}'s result to bind to its free variables. On the other hand, f_{e} is unchanged:

$$f = \lambda^{\mathsf{s}} x{:}|\sigma|^{\mathsf{s}}.e \implies \begin{aligned} f_{\mathsf{e}} &= \lambda^{\mathsf{d}} x_{\mathsf{d}}{:}|\sigma|^{\mathsf{d}}.\mathsf{let}\ x = \uparrow^{\sigma} x_{\mathsf{d}}\ \mathsf{in}\ e_{\mathsf{e}} \\ f_{\mathsf{i}} &= \lambda^{\mathsf{s}} x{:}|\sigma|^{\mathsf{s}}.\mathsf{let}\ \langle fv \rangle^{\mathsf{d}} = \pi_2^{\mathsf{d}}(@^{\mathsf{d}} f_{\mathsf{e}}\,(\downarrow^{\sigma} x))\ \mathsf{in}\ e_{\mathsf{i}} \end{aligned}$$

Since f_{e} now returns the original result in its first field, we can use it directly almost as is to build f_{d} and we can of course still use f_{i} as f_{s}:

$$f_{\mathsf{s}} = f_{\mathsf{i}} \qquad\qquad f_{\mathsf{d}} = \pi_1 \circ f_{\mathsf{e}} = \lambda^{\mathsf{d}} x_{\mathsf{d}}{:}|\sigma_1|^{\mathsf{d}}.\pi_1^{\mathsf{d}}(@^{\mathsf{d}} f_{\mathsf{e}}\, x_{\mathsf{d}})$$

Properties

To define and show correctness of the splitting algorithm, we need an extended notion of equivalence that applies to expressions rather than just values:

$$e \simeq e' \text{ if and only if}$$
$$\text{for any } E \text{ such that } \cdot;\cdot \vdash E[e] : \sigma \text{ then } \cdot;\cdot \vdash E[e'] : \sigma \text{ and } \mathcal{R}_{\mathsf{e}}(E[e]) \simeq \mathcal{R}_{\mathsf{e}}(E[e'])$$

Splitting turns an expression e into a dynamic part E_{e} and a static part e_{i}. The following theorems state that combining E_{e} and e_{i} yields a well-typed term that is semantically equivalent to e.

Theorem 3.4.1 (Type preservation)
if $\Delta;\Gamma \vdash e : |\sigma|^s$ *and* $\Delta;\Gamma \vdash^{split} [\![e]\!] \overset{\sigma}{\Longrightarrow} E_e\,;\, e_i$ *and* $fv = \mathsf{fv}(e_i) - \mathsf{dom}(\Gamma)$ *then* $\Delta;\Gamma \vdash \mathsf{let}\ \langle fv\rangle^d = \pi_2^d(E_{\mathsf{e}}[\langle fv\rangle^d])\ \mathsf{in}\ e_i : |\sigma|^s$.

Theorem 3.4.2 (Correctness)
if $\Delta;\Gamma \vdash e : |\sigma|^s$ *and* $\Delta;\Gamma \vdash^{split} [\![e]\!] \overset{\sigma}{\Longrightarrow} E_e\,;\, e_i$ *and* $fv = \mathsf{fv}(e_i) - \mathsf{dom}(\Gamma)$ *then* $e \;\simeq\; \mathsf{let}\ \langle fv\rangle^d = \pi_2^d(E_{\mathsf{e}}[\langle fv\rangle^d])\ \mathsf{in}\ e_i$.

Both theorems can be proved via induction on the splitting derivation with the help of an invariant. For correctness, the invariant is:

$$\text{For any term } e' \text{ and set of variables } xs$$
$$\text{such that } (\mathsf{fv}(e') - \mathsf{fv}(E_{\mathsf{e}}[e'])) \subseteq xs \subseteq \mathsf{fv}(e')$$
$$\text{then } E_{\mathsf{e}}[e'] \;\simeq\; \mathsf{let}\ \langle xs\rangle^{\mathsf{d}} = \pi_2^{\mathsf{d}}(E_{\mathsf{e}}[\langle xs\rangle^{\mathsf{d}}])\ \mathsf{in}\ e'$$

The invariant for type preservation is similar.

$$t_{\mathsf{co}} = t_{\mathsf{d}} \xrightarrow{\mathsf{s}} t_{\mathsf{s}} \times^{\mathsf{s}} t_{\mathsf{s}} \xrightarrow{\mathsf{s}} t_{\mathsf{d}}$$

$$\begin{aligned}
|\mathsf{int}|^{\mathsf{s}} &= \mathsf{int} \\
|t|^{\mathsf{s}} &= t_{\mathsf{s}} \\
|\tau_1 \to \tau_2|^{\mathsf{s}} &= |\tau_1|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\tau_2|^{\mathsf{s}} \times^{\mathsf{s}} |\tau_1 \to \tau_2|^{\mathsf{d}} \\
|\mathsf{V}(\tau)|^{\mathsf{s}} &= |\tau|^{\mathsf{s}} \\
|\sigma_1 \times \cdots \times \sigma_n|^{\mathsf{s}} &= |\sigma_1|^{\mathsf{s}} \times^{\mathsf{s}} \cdots \times |\sigma_n|^{\mathsf{s}} \times^{\mathsf{s}} |\sigma_1 \times \cdots \times \sigma_n|^{\mathsf{d}} \\
|\sigma_1 \to \sigma_2|^{\mathsf{s}} &= |\sigma_1|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\sigma_2|^{\mathsf{s}} \times^{\mathsf{s}} |\sigma_1 \to \sigma_2|^{\mathsf{d}} \\
|\forall t.\sigma|^{\mathsf{s}} &= \forall^{\mathsf{s}} t_{\mathsf{s}}{:}\mathsf{s}.\forall^{\mathsf{s}} t_{\mathsf{d}}{:}\mathsf{d}.t_{\mathsf{co}} \xrightarrow{\mathsf{s}} |\sigma|^{\mathsf{s}} \times^{\mathsf{s}} |\forall t.\sigma|^{\mathsf{d}}
\end{aligned}$$

$$\begin{aligned}
|\mathsf{int}|^{\mathsf{d}} &= \mathsf{int} \\
|t|^{\mathsf{d}} &= t_{\mathsf{d}} \\
|\tau_1 \to \tau_2|^{\mathsf{d}} &= |\tau_1|^{\mathsf{d}} \xrightarrow{\mathsf{d}} |\tau_2|^{\mathsf{d}} \\
|\mathsf{V}(\tau)|^{\mathsf{d}} &= |\tau|^{\mathsf{d}} \\
|\sigma_1 \times \cdots \times \sigma_n|^{\mathsf{d}} &= |\sigma_1|^{\mathsf{d}} \times^{\mathsf{d}} \cdots \times |\sigma_n|^{\mathsf{d}} \\
|\sigma_1 \to \sigma_2|^{\mathsf{d}} &= |\sigma_1|^{\mathsf{d}} \xrightarrow{\mathsf{d}} |\sigma_2|^{\mathsf{d}} \\
|\forall t.\sigma|^{\mathsf{d}} &= \forall^{\mathsf{d}} t{:}\mathsf{d}.|\sigma|^{\mathsf{d}}
\end{aligned}$$

$$\begin{aligned}
|\,\Delta;\Gamma;\Sigma\,| \;=\; & \Delta';\Gamma' \\
& \text{where } \Delta' = \{t_{\mathsf{s}}{:}\mathsf{s}\,,\; t_{\mathsf{d}}{:}\mathsf{d} \mid t \in \mathsf{dom}(\Delta)\} \\
& \quad \text{and } \Gamma' = \{x{:}|\Gamma(x)|^{\mathsf{s}} \mid x \in \mathsf{dom}(\Gamma)\} \cup \{x_t{:}t_{\mathsf{co}} \mid t \in \mathsf{dom}(\Delta)\} \\
& \qquad\qquad \cup \{z{:}|\tau|^{b} \mid \tau = \Gamma(z) \text{ and } b = \Sigma(z)\}
\end{aligned}$$

Figure 3.13: Type and environment translation.

The splitting algorithm also satisfies the following property:

Theorem 3.4.3 (Static closure)

if $\Delta;\Gamma \vdash e : |\sigma|^{\mathsf{s}}$ and $\Delta;\Gamma \vdash^{\mathit{split}} [\![e]\!] \overset{\sigma}{\Longrightarrow} E_{\mathsf{e}}\,;\, e_{\mathsf{i}}$ then all free variables in e_{i} are either bound in Γ or they have dynamic type (in the context of E_{e}).

We prove this property by inspection of the rules: if a variable is free in e_{i} but not in Γ it can only be because the *sp-share* rule was used, but that rule only applies to dynamic variables or variables which are not free in e_{i}.

Static closure implies that e_{i} contains all the inlinable sub-terms in e so splitting does not hide any inlining opportunities. In other words, when doing partial evaluation of a term containing e, we can substitute e_{i} for e without preventing any reduction (except reductions internal to the terms omitted in e_{i}, obviously). This in turn implies that compilation-unit boundaries have no influence on whether or not a core function gets inlined at a particular call site. We call it the *completeness* property.

3.4.4 The main algorithm

The main algorithm is the translation of the module language, which works similarly to (and uses) the core translation presented earlier, but is interleaved with the splitting algorithm.

$$\boxed{\downarrow^{\sigma}\!:\ |\sigma|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\sigma|^{\mathsf{d}} \quad \text{and} \quad \uparrow^{\sigma}\!:\ |\sigma|^{\mathsf{d}} \xrightarrow{\mathsf{s}} |\sigma|^{\mathsf{s}}}$$

$$\begin{array}{llll}
\downarrow^{\mathsf{int}}x & = x & \uparrow^{\mathsf{int}}x & = x \\
\downarrow^{t}x & = @^{\mathsf{s}}(\pi_2^{\mathsf{s}}x_t)\,x & \uparrow^{t}x & = @^{\mathsf{s}}(\pi_1^{\mathsf{s}}x_t)\,x \\
\downarrow^{\tau_1\to\tau_2}x & = \pi_2^{\mathsf{s}}x & \uparrow^{\tau_1\to\tau_2}x & = \langle \lambda^{\mathsf{s}}x_1{:}|\tau_1|^{\mathsf{s}}.\ \uparrow^{\tau_2}(@^{\mathsf{d}}x\,(\downarrow^{\tau_1}x_1)), x\rangle^{\mathsf{s}} \\
\downarrow^{\mathsf{V}(\tau)}x & = \downarrow^{\tau}x & \uparrow^{\mathsf{V}(\tau)}x & = \uparrow^{\tau}x \\
\downarrow^{\sigma_1\times\cdots\times\sigma_n}x & = \pi_2^{\mathsf{s}}x & \uparrow^{\sigma_1\times\cdots\times\sigma_n}x & = \langle\langle \uparrow^{\sigma_1}(\pi_1^{\mathsf{d}}x),\ldots,\uparrow^{\sigma_n}(\pi_n^{\mathsf{d}}x)\rangle^{\mathsf{s}}, x\rangle^{\mathsf{s}} \\
\downarrow^{\sigma_1\to\sigma_2}x & = \pi_2^{\mathsf{s}}x & \uparrow^{\sigma_1\to\sigma_2}x & = \langle \lambda^{\mathsf{s}}x_1{:}|\sigma_1|^{\mathsf{s}}.\ \uparrow^{\sigma_2}(@^{\mathsf{d}}x\,(\downarrow^{\sigma_1}x_1)), x\rangle^{\mathsf{s}} \\
\downarrow^{\forall t.\sigma}x & = \pi_2^{\mathsf{s}}x & \uparrow^{\forall t.\sigma}x & = \langle \Lambda^{\mathsf{s}}t_{\mathsf{s}}{:}\mathsf{s}.\Lambda^{\mathsf{s}}t_{\mathsf{d}}{:}\mathsf{d}.\lambda^{\mathsf{s}}x_t{:}t_{\mathsf{co}}.\ \uparrow^{\sigma}(x[t_{\mathsf{d}}]^{\mathsf{d}}), x\rangle^{\mathsf{s}}
\end{array}$$

Figure 3.14: Binding-time coercions.

It also relies on the use of pairs that keep both a dynamic and a static version of every module-level value to avoid unnecessary code duplication.

Figures 3.13 and 3.14 extend the type translations $|\cdot|^{\mathsf{s}}$ and $|\cdot|^{\mathsf{d}}$ and the coercions $\downarrow$ and $\uparrow$ to the module calculus. The main change is the case for the type abstraction which we will explain later.

The full staging algorithm is shown in Fig. 3.15. The judgment $\Delta;\Gamma \vdash [\![m:\sigma]\!] \overset{\mathsf{m}}{\Rightarrow} e$ means that under the environments Δ and Γ, the SRC module m of type σ is translated into the TLC term e. Most rules are straightforward. The translation of expressions $\iota_{\mathsf{V}}(c)$ is delegated to the core translation.

The case for module-level function and type abstraction are most interesting. The translation of a module-level function $\lambda x{:}\sigma.m$ begins by recursively translating the body m, and then splitting it. This is done with the judgment $\Delta;\Gamma \vdash [\![m:\sigma]\!] \Longrightarrow e_{\mathsf{e}}\,;\,E_{\mathsf{i}}$. We then build a pair $f = \langle f_{\mathsf{d}}, f_{\mathsf{s}}\rangle^{\mathsf{s}}$ as described in the previous section. The translation of a type abstraction $\Lambda t.m$ follows the same pattern, except for a subtle complication introduced by typing problems discussed below.

Since all module-level code is considered static, it is tempting to think that we do not need pairing at all and can simply represent module entities with the static counterpart. But the e_{e} component obtained from λ-split is dynamic and we thus need coercions to interact with it: both the *sp-var* rule of λ-split (see Fig. 3.12) and the construction of f_{i} out of e_{i} (see Sec. 3.4.3) introduce coercions. And since modules tend to be larger than core

$\boxed{\Delta;\Gamma \vdash [\![m:\sigma]\!] \overset{\mathsf{m}}{\Rightarrow} e}$ such that $|\,\Delta;\Gamma;\cdot\,| \vdash e : |\sigma|^{\mathsf{s}}$

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash [\![x:\Gamma(x)]\!] \overset{\mathsf{m}}{\Rightarrow} x} \qquad \frac{\Delta;\Gamma;\cdot \vdash [\![c:\tau]\!] \overset{\mathsf{s}}{\Rightarrow} e}{\Delta;\Gamma \vdash [\![\iota_{\mathsf{v}}(c):\mathsf{V}(\tau)]\!] \overset{\mathsf{m}}{\Rightarrow} e}$$

$$\frac{\Delta;\Gamma \vdash x_i : \sigma_i \;\; (1 \le i \le n) \qquad \sigma = \sigma_1 \times \cdots \times \sigma_n}{\Delta;\Gamma \vdash [\![\langle x_1,\ldots,x_n\rangle : \sigma]\!] \overset{\mathsf{m}}{\Rightarrow} \begin{array}{l}\mathsf{let}\; x_{\mathsf{s}} = \langle x_1,\ldots,x_n\rangle^{\mathsf{s}} \\ \quad x_{\mathsf{d}} = \langle \downarrow^{\sigma_1} x_1,\ldots,\downarrow^{\sigma_n} x_n\rangle^{\mathsf{d}} \\ \mathsf{in}\; \langle x_{\mathsf{s}}, x_{\mathsf{d}}\rangle^{\mathsf{s}}\end{array}}$$

$$\frac{\Delta;\Gamma \vdash x : \sigma_1 \times \cdots \times \sigma_n \quad 1 \le i \le n}{\Delta;\Gamma \vdash [\![\pi_i x : \sigma_i]\!] \overset{\mathsf{m}}{\Rightarrow} \pi_i^{\mathsf{s}}(\pi_1^{\mathsf{s}} x)} \qquad \frac{\Delta;\Gamma \vdash x_1 : \sigma_2 \to \sigma_1 \quad \Delta;\Gamma \vdash x_2 : \sigma_2}{\Delta;\Gamma \vdash [\![@x_1\, x_2 : \sigma_2]\!] \overset{\mathsf{m}}{\Rightarrow} @^{\mathsf{s}}(\pi_1^{\mathsf{s}} x_1)\, x_2}$$

$$\frac{\Delta;\Gamma x,\sigma_1 \vdash [\![m:\sigma_2]\!] \Longrightarrow e_{\mathsf{e}}\,;\,E_{\mathsf{i}}}{\begin{array}{l}\Delta;\Gamma \vdash [\![\lambda x{:}\sigma_1.m : \sigma_1 \to \sigma_2]\!] \overset{\mathsf{m}}{\Rightarrow} \\ \mathsf{let}\; x_{\mathsf{e}} = \lambda^{\mathsf{d}} x_{\mathsf{d}}{:}|\sigma_1|^{\mathsf{d}}.\;\mathsf{let}\; x = \uparrow^{\sigma_1} x_{\mathsf{d}}\;\mathsf{in}\; e_{\mathsf{e}} \\ \quad x_{\mathsf{i}} = \lambda^{\mathsf{s}} x{:}|\sigma_1|^{\mathsf{s}}.\; E_{\mathsf{i}}[\pi_2^{\mathsf{d}}(@^{\mathsf{d}} x_{\mathsf{e}}\,(\downarrow^{\sigma_1} x))] \\ \mathsf{in}\; \langle x_{\mathsf{i}}, \lambda^{\mathsf{d}} x_{\mathsf{d}}{:}|\sigma_1|^{\mathsf{d}}.\pi_1^{\mathsf{d}}(@^{\mathsf{d}} x_{\mathsf{e}}\, x_{\mathsf{d}})\rangle^{\mathsf{s}}\end{array}} \qquad \frac{\Delta t,;\Gamma \vdash [\![m:\sigma]\!] \Longrightarrow e_{\mathsf{e}}\,;\,E_{\mathsf{i}}}{\begin{array}{l}\Delta;\Gamma \vdash [\![\Lambda t.m : \forall t.\sigma]\!] \overset{\mathsf{m}}{\Rightarrow} \\ \mathsf{let}\; x_{\mathsf{e}} = \Lambda^{\mathsf{d}} t{:}\mathsf{d}.\;\{t/t_{\mathsf{d}}, t/t_{\mathsf{s}}, \langle \mathsf{id}_t^{\mathsf{s}}, \mathsf{id}_t^{\mathsf{s}}\rangle^{\mathsf{s}}/x_t\} e_{\mathsf{e}} \\ \quad x_{\mathsf{i}} = \Lambda^{\mathsf{s}} t_{\mathsf{s}}{:}\mathsf{s}.\Lambda^{\mathsf{s}} t_{\mathsf{d}}{:}\mathsf{d}.\lambda^{\mathsf{s}} x_t{:}t_{\mathsf{co}}.\; E_{\mathsf{i}}[\pi_2^{\mathsf{d}}(x_{\mathsf{e}}[t_{\mathsf{d}}]^{\mathsf{d}})] \\ \mathsf{in}\; \langle x_{\mathsf{i}}, \Lambda^{\mathsf{d}} t{:}\mathsf{d}.\pi_1^{\mathsf{d}}(x_{\mathsf{e}}[t]^{\mathsf{d}})\rangle^{\mathsf{s}}\end{array}}$$

$$\frac{\Delta;\Gamma \vdash x : \forall t.\sigma}{\Delta;\Gamma \vdash [\![x[\tau] : \{\tau/t\}\sigma]\!] \overset{\mathsf{m}}{\Rightarrow} @^{\mathsf{s}}(\pi_1^{\mathsf{s}} x)[|\tau|^{\mathsf{s}}]^{\mathsf{s}}[|\tau|^{\mathsf{d}}]^{\mathsf{s}}\,\langle \downarrow^{\tau}, \uparrow^{\tau}\rangle}$$

$$\frac{\Delta;\Gamma \vdash [\![m_1:\sigma_1]\!] \overset{\mathsf{m}}{\Rightarrow} e_1 \quad \Delta;\Gamma x,\sigma_1 \vdash [\![m_2:\sigma_2]\!] \overset{\mathsf{m}}{\Rightarrow} e_2}{\Delta;\Gamma \vdash [\![\mathsf{let}\; x = m_1\;\mathsf{in}\; m_2 : \sigma_2]\!] \overset{\mathsf{m}}{\Rightarrow} \mathsf{let}\; x = e_1\;\mathsf{in}\; e_2}$$

$\boxed{\Delta;\Gamma \vdash [\![m:\sigma]\!] \Longrightarrow e_{\mathsf{e}}\,;\,E_{\mathsf{i}}}$
such that $|\,\Delta;\Gamma;\cdot\,| \vdash E_{\mathsf{i}}[\pi_2^{\mathsf{d}} e_{\mathsf{e}}] : |\sigma|^{\mathsf{s}}$

$$\frac{\begin{array}{cc}\Delta;\Gamma \vdash [\![m:\sigma]\!] \overset{\mathsf{m}}{\Rightarrow} e & |\,\Delta;\Gamma;\cdot\,| \vdash^{\mathsf{split}} [\![e]\!] \overset{\sigma}{\Longrightarrow} E_{\mathsf{e}}\,;\,e_{\mathsf{i}} \\ fv = \mathsf{fv}(e_{\mathsf{i}}) - \mathsf{dom}(\Gamma) & E_{\mathsf{i}} = (\mathsf{let}\;\langle fv\rangle^{\mathsf{d}} = \bullet\;\mathsf{in}\; e_{\mathsf{i}})\end{array}}{\Delta;\Gamma \vdash [\![m:\sigma]\!] \Longrightarrow E_{\mathsf{e}}[\langle fv\rangle^{\mathsf{d}}]\,;\,E_{\mathsf{i}}}$$

Figure 3.15: Module code translation. id_t^b is a shorthand for $\lambda^b x{:}t.x$.

functions, it is even more important to avoid spurious copies.

Type abstractions As mentioned above, type abstractions introduce some complications. The problem appears when we try to define coercion functions. The naive approach would look like:

$$\downarrow^{\forall t.\sigma} x = \Lambda^{\mathsf{d}} t{:}\mathsf{d}.\ \downarrow^{\sigma}(x[t]^{\mathsf{s}})$$
$$\uparrow^{\forall t.\sigma} x = \Lambda^{\mathsf{s}} t{:}\mathsf{s}.\ \uparrow^{\sigma}(x[t]^{\mathsf{d}})$$

but this is not type correct, since t in the second rule can be static and hence cannot be passed to the dynamic x. Obviously, we need here the same kind of (contra-variant) argument coercion as we use on functions, but our language does not provide us with any way to create a $\downarrow$ operator to apply to types.

Furthermore, the two inner coercions $\downarrow^{\sigma}$ and $\uparrow^{\sigma}$ are not very well defined since σ can have a free variable t. This begs the question: what should $\uparrow^{t}$ do ?

There are several ways to solve these two problems:

- Give up on static type arguments and force any type-variable to be dynamic. This restriction is fairly minor in practice. It only manifests itself when a function is manipulated as data by polymorphic code, such as when a function is passed to the identity function id: the function returned by $@\mathsf{id}\, f$ cannot be inlined even if f is.

- Extend our language with a more powerful type-system that allows intensional type-analysis [HM95]. This seems possible, but would complicate the type-system considerably and potentially the staging and the coercions as well.

- Use a dictionary-passing approach [WB89]: instead of trying to coerce our static t into a dynamic t, we can simply always provide both versions t_{s} and t_{d} along with both $\downarrow^{t}$ and $\uparrow^{t}$ so that the coercions are constructed at the type application site, where t is statically known.

The first solution is simple and effective, but we opted for the third alternative because it has fewer limitations. The static version of $\Lambda t.m$ (before pairing with its dynamic counterpart) looks like:

$$\Lambda^{\mathsf{s}} t_{\mathsf{s}}{:}\mathsf{s}.\Lambda^{\mathsf{s}} t_{\mathsf{d}}{:}\mathsf{d}.\lambda^{\mathsf{s}} x_t{:}t_{\mathsf{co}}.e$$

This means that for every t in the SRC Δ environment, we now have two corresponding type variables t_{s} and t_{d} plus one value variable x_t which holds the two coercion functions $\downarrow^t$ and $\uparrow^t$ As can be seen in Fig. 3.13 (which refines Fig. 3.10) where t_{co} is also defined. This notation is used for convenience in all the figures.

Such an encoding might look convoluted and cumbersome, but type abstractions only represent a small fraction of the total code size and the run-time code size is unaffected, so it is a small price to pay in exchange for the ability to inline code that had to pass through a function like id.

Theorem 3.4.4 (Type preservation)

If $\Delta;\Gamma \vdash m : \sigma$ *and* $\Delta;\Gamma \vdash [\![m{:}\sigma]\!] \Longrightarrow e_{\mathsf{e}}\,;\, E_i$ *then* $|\,\Delta;\Gamma;\cdot\,| \vdash E_i[\pi_2^{d} e_{\mathsf{e}}] : |\sigma|^{s}$.

Together with the residualization theorem 3.3.1, this means that after specialization all the inlinable code has been inlined away.

3.5 Handling the top level

The above presentation only explains how to translate each SRC compilation unit into its TLC counterpart. This section describes in detail how to handle top-level issues to link multiple compilation units together.

Handling of compilation units is not difficult, but is worth looking at not only to get a better idea of how the code flows through the compiler, but also because the treatment of side-effects depends on the specifics of the evaluation of each compilation unit (see Sec. 3.6.1).

As can be seen in Fig. 3.1, we apply λ-split twice. This derives from the need to handle the top-level of the compilation unit in a special way where splitting internal module functions should be done early, while splitting the top-level should be done late. Here is a slightly more detailed diagram:

$$\begin{array}{ccccccccc} & & & sum_{1..n} & & & & & sum \\ & & & \searrow & & & & \nearrow & \\ PRG_{\mathsf{S}} & \xrightarrow{\mathsf{stage+split}} & PRG_1 & \xrightarrow{\oplus} & PRG_2 & \xrightarrow{\mathcal{P}_{\mathsf{e}}} & PRG_3 & \xrightarrow{\mathsf{split}} & PRG_4 \end{array}$$

Instead of spreading the split into two parts, we could of course do it once and for all at the very beginning, but then we would lose the opportunity to move into the export summary copies of wrapper functions (used e.g. for uncurrying, unboxing or flattening) introduced by the intermediate optimization phases.

Doing the split in two steps also forces us to apply the split to TLC terms (it would be silly to have two splitting algorithms). This also motivates our choice to interleave the staging and splitting since the splitting algorithm needs to know which functions are modules and which are not because splitting core functions is often detrimental to performance.

The top-level also gets a special treatment because of separate compilation. A compilation-unit can contain free variables, which are essentially the imports of the unit. Instead of considering such an open term, we close it by turning it into a function from its imports to its exports. More specifically, a compilation-unit in SRC will look like:

$$PRG_{\mathsf{S}} = \lambda\langle imp_1,...,imp_n\rangle{:}\sigma_1 \times \cdots \times \sigma_n.m$$

The translation to TLC assumes that the function is static as well as the imports (these will be import summaries, which are by essence inlinable, after all) and simply translates the body using:

$$\cdot;\{imp_i{:}|\sigma_i|^{\mathsf{s}} \mid 1 \leq i \leq n\} \vdash [\![m:\sigma]\!] \overset{\mathsf{m}}{\Rightarrow} e$$

This recursively splits each and every internal module-level function (the recursion is done by the staging part of the translation which calls λ-split when needed, see the rules for λ and Λ in Fig. 3.15), but leaves the top-level function alone. These internal splits are necessary to allow inlining across module boundaries but still within a compilation unit.

The program now looks like:

$$PRG_1 = \lambda^{\mathsf{s}}\langle imp_1,\ldots, imp_n\rangle^{\mathsf{s}}{:}|\sigma_1|^{\mathsf{s}} \times^{\mathsf{s}} \cdots \times |\sigma_n|^{\mathsf{s}}.e$$

The next step is to bring in copies of the import summaries. Every imp_i has a corresponding summary sum_i generated when that import was compiled. Summaries are the e_{i} half of a split and thus contain free dynamic variables. So we replace each imp_i argument with a copy of sum_i, and add the corresponding new free variables imp_{ij} as new arguments:

$$\begin{aligned}PRG_2 = \lambda^{\mathsf{s}}\langle imp_{11},\ldots, imp_{nk}\rangle^{\mathsf{d}}{:}\sigma_{11} \times^{\mathsf{d}} \cdots \times \sigma_{nk}.&\\ \mathsf{let}\ imp_1 = sum_1&\\ \ldots&\\ imp_n = sum_n&\\ \mathsf{in}\ e&\end{aligned}$$

After that comes the actual partial-evaluation and optimization which ends with a term PRG_3 very much like PRG_2 but with an optimized body e_{o} exempt of any static redex:

$$PRG_3 = \mathcal{P}_{\mathsf{e}}(PRG_2) = \lambda^{\mathsf{s}}\langle imp_{11},\ldots, imp_{nk}\rangle^{\mathsf{d}}{:}\sigma_{11} \times^{\mathsf{d}} \cdots \times \sigma_{nk}.\, e_o$$

We then pass it to the second λ-split, along with the SRC σ output type that we remembered from the staging phase:

$$\Delta; \Gamma \vdash^{\mathsf{split}} [\![e_{\mathsf{o}}]\!] \xRightarrow{\sigma} E_{\mathsf{e}}\ ;\ e_{\mathsf{i}}$$

This split gives us a residual program and an export summary sum that will be used as a sum_i next time around:

$$\begin{aligned} sum &= \lambda^{\mathsf{s}}\langle fv\rangle^{\mathsf{d}}{:}\langle\sigma_{fv}\rangle^{\mathsf{d}}.e_{\mathsf{i}} \qquad\qquad \text{where } fv = \mathsf{fv}(e_{\mathsf{i}})\\ PRG_4 &= \lambda^{\mathsf{d}}\langle imp_{11},\ldots, imp_{nk}\rangle^{\mathsf{d}}{:}\sigma_{11} \times^{\mathsf{d}} \cdots \times \sigma_{nk}.E_{\mathsf{e}}[\langle fv\rangle^{\mathsf{d}}]\end{aligned}$$

The export summary sum will be stashed somewhere to be used when a compilation unit wants to import it. As for PRG_4, it continues through the remaining compilation stages down to machine code.

When the program is run, all the compilation units need to be instantiated in the proper order. Once all the imports imp_{ij} of our unit have been built, PRG_4 is run as follows:

$$\langle exp, \langle fv\rangle^{\mathsf{d}}\rangle^{\mathsf{d}} = \mathcal{R}_{\mathsf{e}}(@^{\mathsf{d}} PRG_4\, \langle imp_{11},..., imp_{nk}\rangle^{\mathsf{d}})$$

Here exp is the original exports of this compilation unit. They will be ignored for all but the main compilation unit (unless one of the dependent units was compiled without cross-module inlining, in which case exp will be used by that unit). $\langle fv\rangle^{\mathsf{d}}$ is the set of exports generated by the λ-split and needed for all the compilation units that depend on the current unit and hence imported sum. When running those dependent units, the current $\langle fv\rangle^{\mathsf{d}}$ will then appear as the arguments $imp_{i1} \ldots imp_{ik}$.

Note that PRG_4 will only be run once and for all whereas sum will be evaluated as many times as it is imported by dependent compilation units. Also, PRG_4 is not completely dynamic since the coercion $\downarrow^{\sigma} x$ of the *sp-var* rule (in Fig. 3.12) introduces static redexes; we have to perform another round of partial evaluation on PRG_4 before feeding it to the backend code generator.

3.6 Extensions

In order to model real world inliners faithfully, our translation still needs various additions which we have not explored in depth yet. We present some here along with other potential extensions.

3.6.1 Side effects

Introducing side-effects is mostly straightforward, with just one exception: The *sp-dup* rule in Fig. 3.12 cannot be applied to non-pure terms since it would duplicate their effects. But reverting to the *sp-share* rule (also in Fig. 3.12) for those side-effecting terms is not an

$$
\begin{array}{lll}
(\mathit{type}) & \sigma ::= \ldots \mid \exists^b t\!:\!k.\sigma \\
(\mathit{term}) & e ::= \ldots \mid \mathsf{open}^b\ v\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e \\
(\mathit{value}) & v ::= \ldots \mid \langle t{=}\sigma : \kappa, v : \sigma\rangle^b
\end{array}
$$

$$
\begin{array}{lll}
|\sigma_1 \to \sigma_2|^{\mathsf{s}} & = & \exists^{\mathsf{s}} t\!:\!\mathsf{d}.|\sigma_1|^{\mathsf{s}} \xrightarrow{\mathsf{s}} t \xrightarrow{\mathsf{s}} |\sigma_2|^{\mathsf{s}} \times^{\mathsf{s}} |\sigma_1|^{\mathsf{d}} \xrightarrow{\mathsf{d}} (|\sigma_2|^{\mathsf{d}} \times^{\mathsf{d}} t) \\
\downarrow^{\sigma_1 \to \sigma_2} x & = & \lambda^{\mathsf{d}} x_1{:}|\sigma_1|^{\mathsf{d}}.\mathsf{open}^{\mathsf{s}}\ x\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ \pi^{\mathsf{d}}_1(@^{\mathsf{d}}(\pi^{\mathsf{s}}_2 x)\, x_1) \\
\uparrow^{\sigma_1 \to \sigma_2} x & = & \mathsf{pack}^{\mathsf{s}}(t = \langle\rangle^{\mathsf{d}}{:}\mathsf{d}, \\
& & \quad \mathsf{let}\ x_{\mathsf{s}} = \lambda^{\mathsf{s}} x_1{:}|\sigma_1|^{\mathsf{s}}.\lambda^{\mathsf{s}} x_2{:}t.\ \uparrow^{\sigma_2}(@^{\mathsf{d}} x\ x_1) \\
& & \quad\quad x_{\mathsf{d}} = \lambda^{\mathsf{d}} x_3{:}|\sigma_1|^{\mathsf{d}}.\langle @^{\mathsf{d}} x\ x_3, \langle\rangle^{\mathsf{d}}\rangle^{\mathsf{d}} \\
& & \quad \mathsf{in}\ \langle x_{\mathsf{s}}, x_{\mathsf{d}}\rangle^{\mathsf{s}}\ )
\end{array}
$$

$$
\frac{\Delta;\Gamma \vdash x_1 : \sigma_2 \to \sigma_1 \quad \Delta;\Gamma \vdash x_2 : \sigma_2}{\begin{array}{l} \Delta;\Gamma \vdash [\![@x_1\, x_2 : \sigma_2]\!] \overset{\mathsf{m}}{\Rightarrow} \\ \mathsf{open}^{\mathsf{s}}\ x_1\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in} \\ \mathsf{let}\ x_{\mathsf{e}} = @^{\mathsf{d}}(\pi^{\mathsf{s}}_2 x)\,(\downarrow^{\sigma_1} x_2) \\ \mathsf{in}\ @^{\mathsf{s}}(@^{\mathsf{s}}(\pi^{\mathsf{s}}_1 x)\, x_2)\,(\pi^{\mathsf{d}}_2 x_{\mathsf{e}}) \end{array}}
\qquad
\frac{\Delta;\Gamma x, \sigma_1 \vdash [\![m : \sigma_2]\!] \Longrightarrow e_{\mathsf{e}}\ ;\ E_{\mathsf{i}}}{\begin{array}{l} \Delta;\Gamma \vdash [\![\lambda x{:}\sigma_1.m : \sigma_1 \to \sigma_2]\!] \overset{\mathsf{m}}{\Rightarrow} \\ \mathsf{pack}^{\mathsf{s}}(t = \sigma_{\mathit{fv}}{:}\mathsf{d}, \\ \quad \mathsf{let}\ x_{\mathsf{e}} = \lambda^{\mathsf{d}} x_{\mathsf{d}}{:}|\sigma_1|^{\mathsf{d}}.\ \mathsf{let}\ x = \uparrow^{\sigma_1} x_{\mathsf{d}}\ \mathsf{in}\ e_{\mathsf{e}} \\ \quad\quad x_{\mathsf{i}} = \lambda^{\mathsf{s}} x{:}|\sigma_1|^{\mathsf{s}}.\lambda^{\mathsf{s}} y{:}t.\ E_{\mathsf{i}}[y] \\ \quad \mathsf{in}\ \langle x_{\mathsf{i}}, x_{\mathsf{e}}\rangle^{\mathsf{s}}\ ) \end{array}}
$$

Figure 3.16: New rules using existential types.

The type, kind, and evaluation semantics should be extended correspondingly. Furthermore, the last rule needs the type σ_{fv} of the free variables of e_{i} which can easily be propagated during splitting.

option either because we would then lose the completeness property that we are looking for. An alternative is to use the following *sp-move* rule that moves the binding to e_{i} instead of merely copying it:

$$
\frac{\Delta;\Gamma \vdash e_1 : \sigma_1 \qquad \Delta;\Gamma x, \sigma_1 \vdash^{\mathsf{split}} [\![e_2]\!] \overset{\sigma}{\Longrightarrow} E_{\mathsf{e}}\ ;\ e_{\mathsf{i}}}{\begin{array}{c} \Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] \overset{\sigma}{\Longrightarrow} \\ \mathsf{let}\ x_1 = \lambda^{\mathsf{d}} x{:}\sigma_1.E_{\mathsf{e}}[\langle \mathit{fv}\rangle^{\mathsf{d}}]\ \mathsf{in}\ \bullet\ ; \\ \mathsf{let}\ x = e_1\ \mathsf{in}\ \mathsf{let}\ \langle \mathit{fv}\rangle^{\mathsf{d}} = @^{\mathsf{d}} x_1\,(\downarrow^{\sigma_1} x)\ \mathsf{in}\ e_{\mathsf{i}} \end{array}}
$$

As presented, this rule is not quite correct because the coercion $\downarrow^{\sigma_1}$ assumes σ_1 is an SRC type but σ_1 is really a TLC type. This can be easily resolved by passing all the SRC types during the translation. Alternatively, we could represent $|\mathsf{int}|^{\mathsf{s}}$ as $\mathsf{int} \times^{\mathsf{s}} \mathsf{int}$, then the coercion $\downarrow$ simply becomes π^{s}_2 for all types.

A potential issue is that, as can be seen in Sec. 3.5, the top-level E_{e} is evaluated once and for all in a global environment, while e_{i} will be evaluated each time it is imported into a client unit. This means that the top-level e_{i} must be free of side-effects. Luckily, we can show that this problem does not appear: the compilation unit does not contain any static free variables or static redexes; so the only static code to split into e_{i} is composed exclusively of values, which have no side effects.

This *sp-move* rule could also be used for other purposes, such as specializing a binding in the context of the client, as was suggested briefly in Blume and Appel's paper (they did not have such a rule).

Another approach altogether to the handling of side-effects is to notice that since e_{i} has to be pure, we can turn $@^{\mathsf{s}}x_1\,x_2$ into $@^{\mathsf{d}}(@^{\mathsf{s}}x_{\mathsf{i}}\,x_2)\,(\pi_2^{\mathsf{d}}(@^{\mathsf{d}}x_{\mathsf{e}}\,(\downarrow x_2)))$ and then split out the remaining static application which is known to be pure. We do not even need to change the splitting algorithm itself, but just two rules in Fig. 3.15.

The key is that we can do this rewrite even if we do not know x_1. We simply need to represent x_1 as a pair $\langle x_{\mathsf{i}}, x_{\mathsf{e}}\rangle^{\mathsf{s}}$. But of course, this is already the case for other reasons, so the changes are very minor.

Of course, there is a catch: the type of the free variables of e_{i} suddenly leak into the type of $|\sigma_1 \to \sigma_2|^{\mathsf{s}}$ which becomes an existential type.

Figure 3.16 shows what the rules would look like. Some of the work is now shifted from the function definition to the function application, but overall, the complexity of terms is not seriously impacted. Apart from the introduction of existential types, this translation variant also requires an impredicative calculus. TLC was already impredicative, so no changes were required there.

3.6.2 Recursion

The TLC calculus lacks fixpoint. Adding recursive functions does not pose any conceptual problem, except for the risk of compilation not terminating and the fact that some of our proofs rely on the fact that the static computation is strongly normalizing. There are several reasonable solutions to these problems either from the inlining community or from the partial evaluation community. The most trivial solution is to allow fixpoint on dynamic terms only, which amounts to disallowing inlining of recursive functions, but since it can be important to allow inlining even in the presence of recursion, one can also do a little bit of analysis to find a conservative estimate of whether or not a risk of infinite recursion is present [PM99].

Recursion on types is more challenging since recursive data-structures cannot (or should not) be coerced. In our case, however, this restriction only applies to coercions from `d` to `s`, so we can always work around the problematic cases by forcing recursive data-structures to be dynamic.

3.6.3 Optimizations as staged computation

With a full λ-calculus available at compile-time, we can now provide facilities similar to macros, or rather to Lisp's compiler macros. For example, a compiler macro for multiplication could test its arguments at compile time and replace the multiplication with some other operations depending on whether or not one of the value is statically known and what value it takes. Using such a facility we could move some of the optimizations built into the compiler into a simple library, making them easily extensible.

A more realistic use in the short term is to encode the predicate for conditional inlining directly into the language. The current inliner allows inlining hints more subtle than the ones present in SRC. They can express a set of conditions that should hold at the call site in order for the call to be inlined. For example, `map` will only be inlined if it is applied to a known function.

We could now strip out those ad-hoc annotations and simply write `map` as a compile-time function that intensionally analyzes its arguments and either returns a copy of its body if the function argument is a λ-expression or returns just a call to the common version if the argument is a variable (i.e. an escaping function).

3.6.4 Staging refinement

Partial evaluation as well as other optimizations will sufficiently change the shape of the code to justify or even require refining the binding-time annotations. This can happen because a function has been optimized down to just a handful of statements, or because it has been split into a wrapper and a main body or any other reason.

Turning a dynamic function into a static one is not very difficult to implement, but more work needs to be done to express it cleanly within our framework.

It seems to require among other things the ability to optimize away pairs of coercions that cancel each other out such as not only $\downarrow^\sigma\uparrow^\sigma x$ (which is trivially done by the partial evaluator) but also $\uparrow^\sigma\downarrow^\sigma x$ which appears to involve evaluation of dynamic code at compile-time.

3.6.5 Link-time optimizations

Another extension is to add multiple levels so that we can express compile-time execution, link-time execution, run-time code generation and more.

This will require extending a calculus such as $\lambda^{\bigcirc}$ [Dav96] with at least some form of polymorphism, but should not pose any real problem, except that the kind of tricks we used to work around the lack of simple coercion for type abstraction might need to be generalized to n-levels. If n is unbounded, it might not be possible and even if it is bounded, it might be impractical.

Also, the use of pairs of fully-static and fully-dynamic representations of the same original expressions would not generalize to n-levels easily, but could still be kept for the benefit of the compile stage.

3.6.6 Implementation

As mentioned in the introduction, this paper was motivated by the need to better understand the behavior of our inliner in SML/NJ. Since our implementation handles the complete SML language in a production compiler, it has to deal with all the issues mentioned above. Here are the most important differences between the model presented in this paper and the actual code:

- For historical reasons, our intermediate language is predicative, which prevents us from using existential packages to solve the problem of side effects. Instead we simply revert to using the *sp-share* rule and lose the completeness property.
- Our language allows recursion both for dynamic and for static functions. Termination of $\mathcal{P}_{\mathsf{e}}(e)$ is ensured by a simple conservative loop detection.
- The dynamic and static pairs we use to avoid spurious code duplication are represented in an ad-hoc way that eliminates the redundancy. This ad-hoc representation looked like a good idea at the time, but made it unnecessarily painful to add the refinement described in Sec. 3.4.3.

3.7 Related work

Functional-language compilers such as O'Caml [Ler95], SML/NJ [App91], GHC [PM99], and TIL [Tar96], all spend great efforts to provide better support to inlining. Although none of them models inlining as staged computation, the heuristics for detecting what functions should be inlined are still useful in our framework. In fact, our FLINT optimizer [MBS99] inherits most of the heuristics used in the original SML/NJ compiler.

Control-flow analysis (CFA) [Shi91, Ash97] is an alternative to λ-splitting to propagate inlining information across functions and functors. It tries to find, for example via abstract interpretation, the set of functions possibly invoked at each call site in the program. It offers the advantage of requiring less code duplication and may expose more opportunities for inlining inside a compilation unit. For example, in a code such as:

$$\mathsf{let}\ f\ x\ y = ..y\ x..\ \mathsf{and}\ g\ x = ...\ \mathsf{in}\ \langle f\ 1\ g, f\ 2\ g\rangle$$

CFA can inline the function g into f without inlining f whereas our inliner will only reach the same result if it can first inline the two calls to f. On the other hand, in a code such as $f\ g\ x$ where f is a functor that ends up returning its argument unchanged, our inliner will be able to replace the code with $g\ x$, no matter how f is defined, whereas in the case of CFA, if the definition of f is sufficiently complex, a costly polyvariant analysis is needed to discover that the code can be replaced with $g\ x$.

Partial evaluation is a very active research area. Jones et al [JGS93] gives a good summary about some of the earlier results. Danvy's paper [Dan96] on type-directed partial evaluation inspired us to look into sophisticated forms of binding-time coercions.

Tempo [CN96, MCB99] is a C compiler that makes extensive use of partial-evaluation technologies. Its main emphasis is however on efficient runtime code generation. Sperber and Thiemann [ST96, ST97] worked on combining compilation with partial evaluation, however they were not concerned with modeling the inlining optimization as done in a production compiler.

Nielson and Nielson [NN92] gave an introduction to a two-level λ-calculus. Davies and Pfenning [DP96] proposed to use modal logic to express staged computation. Moggi [Mog97] pointed out that both of these calculi are subtly different from the two-level calculus used in partial evaluation [JGS93]. Taha et al [Tah99, TS97, MTBS99] showed how to combine these different calculi into a single framework.

Our TLC calculus is an extension of Moggi's two-level $\lambda 2sd$ calculus [Mog97] with the System-F-style polymorphism [Gir72, Rey74]. Davies [Dav96] used the temporal logic to model an n-level calculus which naturally extends $\lambda 2sd$.

Foster et al. [FFA99] proposed to use qualified types to model source-level program directives. Their framework can be applied to binding time annotations but these annotations would have to become parts of the type specifications. Our inlining annotations on the other hand do not change the source-level type specifications.

Blume and Appel [BA97] suggested λ-splitting to support cross module inlining. Their algorithm is based on a weakly-typed λ-calculus and provides a convenient cross-module inlining algorithm. Our work extends theirs by porting their algorithm to a much more powerful language and formalizing it. By using the two-level λ-calculus we can express some of the inliner's behavior in the types.

O'Caml [Ler95] collects the small inlinable functions of a module into its *approximation* and then reads in this extra info (if available) when compiling a client module. It works very well across modules and can even inline functions from within a functor to the client of the functor, but is unable to inline the argument of a functor. For example, passing a module through a trivial "adaptor" functor (which massages a module to adapt it to some other signature, e.g.) will lose the *approximation*, preventing inlining.

By encoding the equivalent of *approximations* directly into types, Shao [Sha98] presents an alternative approach which allows the full inlining information to be completely propagated across functor applications by propagating it along with the types. But this comes at the cost of a further complication of the module elaboration. Another problem is that some of the functions we might want to inline (such as uncurry wrappers) do not yet exist at the time of module elaboration.

Recently, Ganz et al [GST01] presented an expressive, typed language that supports generative macros. The language, MacroML, is defined by an interpretation into MetaML [TS97]. This is similar to our approach because inlinable functions can also be viewed as macros; the translation from MacroML to MetaML resembles our translation from the source calculus SRC to the target calculus TLC. There are, however, several major differences. First, in MacroML, macros and functions are different language constructs; macros never escape so they can be unconditionally marked as "static" and no coercions or polymorphic binding-time annotations are ever needed; in our SRC calculus, however, functions that are marked as inlinable are still treated as regular functions so they can escape in any way they like. Second, it is unclear how MacroML can be extended to support ML-style modules; MacroML assigns different types to macros and functions so exporting macros would require adding new forms of specifications into ML signatures; in our SRC language, however, a function is always assigned the same type whether it is marked as inlinable or not, so we can just reuse the existing ML module language. More annoyingly, macro definitions might need to be included in module signatures, to allow separate compilation. Third, unlike MacroML, we presented various techniques to control code duplication—this is crucial for cross-module inlining since naively expanding every functor in ML would certainly cause code explosion. On the other hand MacroML simply shifts the burden onto the programmer, which has also the advantage of allowing recursive macro definitions, simply declaring infinite macro expansion to be the mistake of the programmer.

3.8 Conclusions and future work

Expressing inlining in terms of a staged computation allows us to better formalize the behavior of the inliner and provide strong guarantees of what gets inlined where.

We have shown how this can be done in the context of a realistic two-level polymorphic language and how it interacts with cross-module inlining. The formalism led us to a clean design in which we can easily show that code is only and always duplicated when useful.

The present design eliminates run-time penalties usually imposed by the powerful abstraction mechanism offered by parameterized modules, enabling a more natural programming style. More importantly, our algorithm provides such flexibility while still maintaining separate compilation.

An interesting question is whether or not using monomorphic staging annotations was a good choice. It seems that polymorphism could allow us to do away with the coercions, although it would at least require the use of continuation passing style in order to maintain precision of annotations.

Chapter 4

Principled scavenging

The correctness of most type-safe systems relies critically on the correctness of an underlying garbage collector (GC). This also holds for Proof-Carrying Code (PCC) [Nec97] and Typed Assembly Languages (TAL) [MWCG98]—both of which aim to minimize the trusted computing base by directly certifying the actual machine code. Unfortunately, these systems cannot get rid of the dependency on a trusted garbage collector. Indeed, constructing a verifiably type-safe garbage collector is widely considered as one of the major open problems in the area of certifying compilation [Mor00, Cra00].

Recently, Wang and Appel [WA01] proposed to tackle the problem by building a tracing garbage collector on top of a region-based calculus. My work builds on theirs but makes the following new significant contributions:

- I show how to use intensional type analysis (ITA) [TSS00, HM95] to accurately describe the contract between the mutator and the collector and how the same framework can be applied to construct various different type-safe GCs.

- Using ITA to typecheck GC may seem to be an obvious idea to some people, however, none of the previous work [WA99, STS00, TSS00] have succeeded in getting it to work. Indeed, Wang and Appel [WA01] subsequently gave up on using ITA. I show why the problem is nontrivial (see Section 4.1.2) and how to modify the basic ITA framework to solve the problem.

- Wang and Appel's collector [WA01] relies on whole-program analysis and code duplication to support higher-order and polymorphic languages; this breaks separate compilation and is impractical. I show how to use runtime type analysis to write my GC as a library (thus no code duplication) and how to directly support higher-order polymorphic functions.

- I expose in detail how to implement and certify efficient forwarding pointers. Making them type-safe is surprisingly subtle (see Section 4.6). Wang and Appel [WA01] also claim to support forwarding pointers but their scheme is less efficient and it is unclear whether it is sound.

- I also show how to handle generations with a simple extension of my base calculus.

- A garbage collector is type-safe only if it is written in a provably type-safe language. I have complete type-soundness proofs for all my calculi. Wang and Appel's collectors [WA01, WA00], on the other hand, are not fully formalized.

Although this work is theoretical in nature, I believe it will be of great interests to the general audience, especially those who are looking to apply new language theory to solve important practical problems such as mobile-code safety and certifying compilation. The next chapter will show how to address the more problematic limitations.

4.1 Motivation and approach

Why do I want a type-safe garbage collector?

The explosive growth of the Internet has induced newfound interest in mobile computation as well as security. Increasingly, applications are being developed at remote sites and then downloaded for execution. A robust mobile code system must allow code from potentially untrusted sources to be executed. At the same time, the system must detect and prevent the execution of malicious code.

The safety of such a system depends not only on the properties of the code being downloaded, but also on the security of the host system itself, or more specifically, its trusted computing base (TCB).

Proof-carrying code and typed assembly languages have been proposed to reduce the size of this TCB by bundling the untrusted code with a mechanically checkable proof of safety, where the safety is usually defined as type-safety. Such systems only need to trust their verifier and runtime system rather than their whole compiler suite.

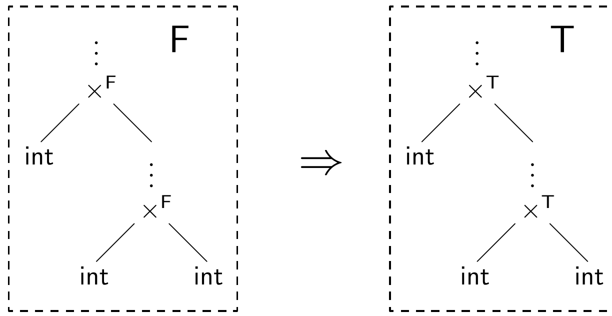
But all these certifying-compiler projects (e.g., PCC, TAL) still crucially rely on the correctness of a tracing garbage collector for their safety. Recently, both Crary [Cra00] and Morrisett [Mor00] have characterized type-safe garbage collection as one of the major open problems in the area of certifying compilation.

A type-safe GC is not only desirable for reducing the size of the TCB but also for making it possible to ship custom-tailored GC along with mobile code, or to choose between many more GC variants without risking the integrity of the system. Writing GC inside a type-safe language itself also makes it possible to achieve principled interoperation between garbage collection and other memory-management mechanisms (e.g., those based on malloc-free and regions). Indeed, the major software-engineering benefit is that a type-safe GC must make explicit the contract between the collector and the mutator and it must make sure that it is always respected. Without typechecking, such rules can prove difficult to implement correctly and bugs can be very difficult to find.

4.1.1 The problem

Recently, Wang and Appel [WA99] proposed to tackle the problem by layering a stop-and-copy tracing garbage collector [Wil92] on top of a region based calculus, thus providing both type safety and completely automatic memory management.

A region calculus [TT94] annotates the type of every heap object with the region in which it is allocated (such as $\sigma_1 \times^\rho \sigma_2$ where ρ is the region), thus allowing to safely reclaim memory by freeing any region that does not appear in any of the currently live types.



$$
\begin{aligned}
&\mathit{copy} : \forall \mathsf{F}.\forall \mathsf{T}.\forall \alpha.(\alpha \to \alpha[{}^{\mathsf{T}}/_{\mathsf{F}}]) \\
&\mathit{GC} = \Lambda \mathsf{F}.\Lambda \alpha.\lambda(x:\alpha, k:\forall \rho.\alpha[{}^{\rho}/_{\mathsf{F}}] \to 0) \\
&\qquad \mathsf{let\ region\ T\ in} \\
&\qquad \mathsf{let}\ y = \mathit{copy}[\mathsf{F}][\mathsf{T}][\alpha](x)\ \mathsf{in} \\
&\qquad \mathsf{only\ T\ in}\ k[\mathsf{T}](y)
\end{aligned}
$$

Figure 4.1: Stop-and-Copy from region F to region T.

GC is written in continuation passing style (CPS). It takes the current region, the heap and a continuation and begins by allocating a new region T with "let region T in e". It then copies the heap into this new region and frees the old region implicitly with "only T in e" which tells that all regions but T can be reclaimed. This way of freeing regions was introduced by Wang and Appel to circumvent problems linked to aliasing of regions.

The basic idea in building a type-safe GC is to concentrate on type-safety rather than correctness. Rather than try to prove that the *copy* function faithfully copies all the heap, I just need to show that it has a type looking somewhat like $\forall \alpha.(\alpha \to (\alpha[\mathsf{T}/\mathsf{F}]))$ where $(\alpha[\mathsf{T}/\mathsf{F}])$ stands for the type α where the region annotation T is substituted for F (see Fig. 4.1.1). Assuming I have such a function and I don't keep any reference to the region F, the region calculus will allow to safely reclaim F.

Clearly, there is no correctness guarantee in sight since the *copy* function might return a completely different value or might not faithfully reproduce the original graph, but it ensures type-safe execution of the whole mutator-collector system and even offers a form of type-preservation guarantee.

The main problem is clearly to write this *copy* function which needs to trace through arbitrary heap structures at runtime. Therefore, the language needs to support some form of runtime type information in order to do the actual *copy*.

In their followup paper [WA01], Wang and Appel suggest to circumvent the problem of runtime type information using a mix of monomorphization and defunctionalization (a form of closure conversion due to Reynolds [Rey72, TO98] which reduces the language to a mono-morphic first-order calculus). However, this approach suffers from several major drawbacks:

- It can introduce a significant code size increase and forces the use of separate specialized *GC* and *copy* functions for each type appearing in the program. Instead of the promised flexibility to choose among various GC variants, this approach locks you into a single 100% tailor-made collector.

- Monomorphization is not applicable in the presence of recursive polymorphism or existential packages, so their type-safe GC cannot handle languages with polymorphic recursion or abstract types.

- Monomorphization and defunctionalization all rely on whole program analysis which are clearly imcompatible with separate compilation.

- Finally, although their type-safe GC does properly formalize the interaction between the mutator and the collector, the formalization is hidden inside the compiler and hence does not allow to bring out open the overly intimate relationship between the GC and the compiler.

They also try to preserve sharing using forwarding pointers. The rough sketch of the solution they propose is similar to the one I developed (which was done independently). It relies mostly on a form of *cast* which allows covariant subtyping of references. Making sure that this cast is sufficiently constrained to be safe is difficult. Their informal presentation is incomplete and possibly incorrect, and leaves many important questions unanswered.

4.1.2 My solution

I want to do away with any form of whole program analysis so as to make the mutator and the collector independent in order to reap the promised benefits of more flexibility and clearer interaction between mutator and GC.

In this chapter, I present a different approach for writing the *copy* function, relying on runtime type analysis. The substitution present in the return type of *copy* as well as the need to observe types at runtime leads one very naturally to use intensional type analysis (ITA) [TSS00, HM95]. In fact, an earlier paper of Wang and Appel [WA99] was titled "safe garbage collection = regions + intensional type analysis" but they subsequently gave up on using ITA and opted for the solution mentioned above [WA01]. Saha et al. [STS00, TSS00] also tried to use ITA to write the *copy* function, but their attempt is missing crucial details and didn't really work either because they failed to properly separate the preserved part of the type from the part that is modified.

Intensional type analysis

A full description of ITA is outside the scope of this thesis, but in short, ITA provides a typecase term to examine at runtime the actual type with which a type variable was instantiated (using some form of runtime type descriptor) and a typerec type to write type functions that do the equivalent of a *fold* to map one type into another. You can thus define a type function that replaces all occurrences of Int with Bool. One important property is that typerec functions only analyse type constructors and linger when faced with a type variable until that variable is instantiated.

A case for symmetry

So what is the problem? It seems that ITA provides us with just the right tools. We can for example write a simple typerec function $\mathsf{S}_{\mathsf{T},\mathsf{F}}(\sigma)$ which substitutes region annotations T for F by recursively analyzing the type σ. At the term level, in the body of *copy*, we can similarly use typecase to determine the actual type of the object we are copying.

But that means that the type grows each time we go through the GC, from σ to $\mathsf{S}_{\mathsf{T},\mathsf{F}}(\sigma)$ to $\mathsf{S}_{\rho,\mathsf{T}}(\mathsf{S}_{\mathsf{T},\mathsf{F}}(\sigma))$ .... This may seem unimportant since S should be reduced away anyway. But $\mathsf{S}_{\rho,\mathsf{T}}(\alpha)$ cannot be reduced further until α is instantiated: $\exists\alpha.\mathsf{S}_{\mathsf{T},\mathsf{F}}(\alpha)$ is a normal form. So the accumulation of S operators is a real problem, since $\mathsf{S}_{\rho,\mathsf{F}}(\alpha)$ is not equal to $\mathsf{S}_{\rho,\mathsf{T}}(\mathsf{S}_{\mathsf{T},\mathsf{F}}(\alpha))$.

We could arrange for $\mathsf{S}_{\rho,\mathsf{T}}(\mathsf{S}_{\mathsf{T},\mathsf{F}}(\sigma))$ to reduce to $\mathsf{S}_{\rho,\mathsf{F}}(\sigma)$. But then all types become $\mathsf{S}_{\rho,\mathsf{F}}(\sigma)$ (where F is the "initial region") except before the first collection. Also it is very ad-hoc and only works if S obeys this kind of idempotency.

A better approach is to ensure that the input and output types of *copy* are symmetric. We first redefine $\mathsf{S}_\rho(\sigma)$ which simply substitutes ρ for any region annotation (why bother with an initial region) and then redefine *copy* to have type $\forall\mathsf{F}.\forall\mathsf{T}.\forall\alpha.(\mathsf{S}_\mathsf{F}(\alpha) \to \mathsf{S}_\mathsf{T}(\alpha))$ which gets us rid of the special case before the first collection and does not require any special reduction rule for S since GC does not increase the size of the type any more.

A case for tags

The above solution looks good until we try to copy existential packages $\exists\alpha\in\Theta.\sigma$, used to encode closures. The Θ annotation is used to bound the set of regions that can appear in the witness type hidden under the type variable α.

Opening an existential package of type $\exists\alpha\in\{\mathsf{F}\}.\mathsf{S}_\mathsf{F}(\alpha)$, gives us the value σ of α (i.e. the witness type) and a value of type $\mathsf{S}_\mathsf{F}(\sigma)$. Recursively applying *copy* to that value will return a new value of type $\mathsf{S}_\mathsf{T}(\sigma)$, but how can we construct the new existential package of type $\exists\alpha\in\{\mathsf{T}\}.\mathsf{S}_\mathsf{T}(\alpha)$? Reusing σ for the witness type will not do since σ is not constrained to $\{\mathsf{T}\}$ but to $\{\mathsf{F}\}$. A witness of $\mathsf{S}_\mathsf{T}(\sigma)$ cannot work either; the only correctly typed package we can produce is $\langle\alpha{=}\mathsf{S}_\mathsf{T}(\sigma), v:\alpha\rangle$ which has type $\exists\alpha\in\{\mathsf{T}\}.\alpha$.

Clearly, we are again trying to push a new S onto the type rather than replacing an S with another. So we can again arrange for $\mathsf{S}_\mathsf{T}(\mathsf{S}_\mathsf{F}(\sigma))$ to reduce to $\mathsf{S}_\mathsf{T}(\sigma)$, but we really do not want to tie our hands with such an ad-hoc and restrictive scheme.

Instead, we can pay a bit more attention to what we do and observe that $\mathsf{S}_\rho(\sigma)$ makes region annotations on σ completely useless, so instead of trying to get those annotations right only to see them substituted we can simply define a parallel set of non-annotated types τ (that we will call *tags*). Since tags have no region annotations, we can hide them in tag variables without any Θ constraint, which side-steps the problem when copying existentials conveniently. Note that contrary to common practice, our tags are not attached to their corresponding objects but are managed completely independently.

Such a split between types and tags is not a new concept since it was already used in the work on intensional type analysis where tags were called *constructors* [HM95, CW99]. But here, tags take on more significance since they correspond to a source-level notion of type and will be mapped to *different* actual types with different type functions M (formerly S) which are used to encapsulate all the constraints that mutator data has to satisfy in order for the collector to do its job. As you will see in sections 4.6 and 4.7 we will use a non-trivial M mapping to force the mutator to provide space for forwarding pointers and to enforce the invariant that references do not point from the old generation to the new.

4.2 Source language λ_{CLOS}

For simplicity of the presentation, the source language we propose to compile and garbage collect is the simply typed λ-calculus.

In order to be able to use our region calculus, we need to convert the source program into a continuation passing style form (CPS). And we also need to close our code to make all data manipulation explicit, so we turn all closures into existential packages.

We will not go into the details of how to do the CPS conversion [DF92] and the closure conversion using existentials [MMH96, MH97].

The language used after CPS conversion and closure conversion is the language λ_{CLOS} shown below.

$$
\begin{array}{lll}
(\mathit{types}) & \tau ::= \mathsf{Int} \mid t \mid \tau_1 \times \tau_2 \mid \tau \to 0 \mid \exists t.\tau \\
(\mathit{values}) & v ::= n \mid f \mid x \mid \langle v_1, v_2 \rangle \mid \langle t{=}\tau_1, v{:}\tau_2 \rangle \\
(\mathit{terms}) & e ::= \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \pi_i v\ \mathsf{in}\ e \mid v_1(v_2) \\
 & \qquad \mid \mathsf{open}\ v\ \mathsf{as}\ \langle t, x \rangle\ \mathsf{in}\ e \mid \mathsf{halt}\ v \\
(\mathit{programs}) & p ::= \mathsf{letrec}\ \overrightarrow{f = \lambda(x{:}\tau).e}\ \mathsf{in}\ e
\end{array}
$$

Since functions are in CPS, they never return, which we represent with the arbitrary return type 0, often pronounced *void*. $\langle v_1, v_2\rangle$ builds a pair while $\pi_i v$ selects its i^{th} element. To represent closures, the language includes existential packages of type $\exists t.\tau_2$ and constructed by $\langle t{=}\tau_1, v{:}\tau_2\rangle$. The construct

$$\mathsf{open}\ v\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e$$

takes an existential package v, binds the witness type to t and the value to x, and then executes e. The complete program

$$\mathsf{letrec}\ \overrightarrow{f_i = \lambda(x_i{:}\tau_i).e_i}\ \mathsf{in}\ e$$

consists of a list of mutually recursive closed function declarations followed by the main term to be executed.

4.3 Target language λ_{GC}

We translate λ_{CLOS} programs into our target language λ_{GC}. The target language is also used to write the garbage collector. Figure 4.2 gives the syntax of λ_{GC}; the semantics are presented in Section 4.5). λ_{GC} extends λ_{CLOS} with regions [TT94] and (fully reflexive) intensional type analysis [TSS00]. Functions are also fully closed and use CPS but they can additionally be polymorphic over tags and regions.

4.3.1 Regions

Our region calculus uses *reference* values $\nu.\ell$ of type $(\sigma\ \mathsf{at}\ \nu)$. ν is the region in which the object is allocated and ℓ is the actual location within that region. Object allocation and memory accesses are made explicit with put and get. In order to trigger GC, ifgc allows us to check whether a region is full. Ensuring timely collection using heap limit checks or other mechanisms is outside the scope of this paper, so put never fails, even if the region is "full".

$$
\begin{array}{lll}
(\mathit{tenv}) & \Delta & ::= \cdot \mid \Delta, t\!:\!\kappa \\
(\mathit{venv}) & \Gamma & ::= \cdot \mid \Gamma, x\!:\!\sigma \\
(\mathit{renv}) & \Theta & ::= \cdot \mid \Theta, \rho \\
(\alpha\ \mathit{env}) & \Phi & ::= \cdot \mid \Phi, \alpha\!:\!\Theta \\
(\mathit{region\ types}) & \Upsilon & ::= \{\ell_1\!:\!\sigma_1, \ldots, \ell_n\!:\!\sigma_n\} \\
(\mathit{mem\ types}) & \Psi & ::= \{\mathsf{cd}\!:\!\Upsilon_{\mathsf{cd}}, \nu_1\!:\!\Upsilon_1, \ldots, \nu_n\!:\!\Upsilon_n\} \\
(\mathit{regions}) & R & ::= \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \\
(\mathit{memories}) & M & ::= \{\mathsf{cd} \mapsto R_{\mathsf{cd}}, \nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\} \\
(\mathit{states}) & P & ::= (M, e) \\
(\mathit{regions}) & \rho & ::= \nu \mid r \\
(\mathit{kinds}) & \kappa & ::= \Omega \mid \Omega \to \Omega \\
(\mathit{tags}) & \tau & ::= t \mid \mathsf{Int} \mid \tau_1 \times \tau_2 \mid \tau \to 0 \mid \exists t.\tau \\
 & & \quad \mid \lambda t.\tau \mid \tau_1\tau_2 \\
(\mathit{types}) & \sigma & ::= \mathsf{int} \mid \sigma_1 \times \sigma_2 \mid \forall[\vec{t\!:\!\kappa}][\vec{r}](\vec{\sigma}) \to 0 \mid \exists t\!:\!\kappa.\sigma \mid \sigma \ \mathsf{at}\ \rho \mid \mathsf{M}_\rho(\tau) \\
 & & \quad \mid \alpha \mid \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0 \mid \exists \alpha\!:\!\Theta.\sigma \\
(\mathit{values}) & v & ::= n \mid x \mid \nu.\ell \mid \langle v_1, v_2 \rangle \mid \langle t\!=\!\tau, v\!:\!\sigma \rangle \mid v[\![\vec{\tau}]\!] \\
 & & \quad \mid \langle \alpha\!:\!\Theta\!=\!\sigma_1, v\!:\!\sigma_2 \rangle \mid \lambda[\vec{t\!:\!\kappa}][\vec{r}](\overrightarrow{x\!:\!\sigma}).e \\
(\mathit{operations}) & \mathit{op} & ::= v \mid \pi_i v \mid \mathsf{put}[\rho]v \mid \mathsf{get}\ v \\
(\mathit{terms}) & e & ::= v[\vec{\tau}][\vec{\rho}](\vec{v}) \mid \mathsf{let}\ x = \mathit{op}\ \mathsf{in}\ e \mid \mathsf{halt}\ v \mid \mathsf{ifgc}\ \rho\ e_1\ e_2 \\
 & & \quad \mid \mathsf{open}\ v\ \mathsf{as}\ \langle t, x \rangle\ \mathsf{in}\ e \mid \mathsf{open}\ v\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{let\ region}\ r\ \mathsf{in}\ e \mid \mathsf{only}\ \Theta\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{tcase}\ \tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1t_2.e_{\times}; t_e.e_{\exists}) \\
(\mathit{tagnf}) & \tau' & ::= t \mid \mathsf{Int} \mid \tau' \to 0 \mid \tau_1' \times \tau_2' \mid \exists t.\tau' \mid \lambda t.\tau' \mid t\tau'
\end{array}
$$

Figure 4.2: Syntax of λ_{GC}

Region allocation and reclamation is done with letregion and only. Deallocation of a region is implicit since only lists the regions that should be kept. This neatly works around aliasing problems, at the cost of a more expensive deallocation operation (only needs to go through the list of all regions to find which ones need to be reclaimed). In our case, we have very few regions and deallocate them only occasionally, so it is a good tradeoff.

4.3.2 Functions and code

Since programs in λ_{GC} are completely closed, we can separate code from data. The memory configuration enforces this by having a separate dedicated region cd for all the code blocks. A value $\lambda[\vec{t}][\vec{r}](\overrightarrow{x\!:\!\sigma}).e$ is only an array of instructions (which can contain references to other values in cd) and needs to be put into a region to get a function pointer before one can call it. In practice, functions are placed into the cd region when translating code from λ_{CLOS} and never directly appear in λ_{GC} code.

The indirection provided by memory references allows us to do away with letrec. For convenience, we will use $\mathsf{fix} f.e$ in examples, but in reality, e will be placed at the location ℓ in the cd region and all occurrences of f will be replaced by $\mathsf{cd}.\ell$.

4.3.3 Intensional type analysis

As explained earlier, we have split the notion of type into two. Tags represent the runtime type descriptors and map very directly to source-level types without any region annotations. The only different between λ_{CLOS} types and λ_{GC} tags is the addition of tag functions $\lambda t.\tau$ and tag applications $\tau_1 \tau_2$, which are needed for type analysis of existentials [TSS00]. To do the actual analysis of tags, terms include a *refining* typecase construct tcase, i.e. a more refined tag is substituted for τ in each arm of the tcase. Finally, instead of a full-blown $\mathsf{typerec}$ construct we only provide a hard-coded M, to keep the presentation simpler. $\mathsf{M}_\rho(\tau)$ is the type corresponding to the tag τ complemented with region annotations ρ:

$$\begin{aligned}
\mathsf{M}_\rho(\mathsf{Int}) &\Longrightarrow \mathsf{int} \\
\mathsf{M}_\rho(\tau_1 \times \tau_2) &\Longrightarrow (\mathsf{M}_\rho(\tau_1) \times \mathsf{M}_\rho(\tau_2)) \text{ at } \rho \\
\mathsf{M}_\rho(\exists t.\tau) &\Longrightarrow (\exists t\!:\!\Omega.\mathsf{M}_\rho(\tau)) \text{ at } \rho \\
\mathsf{M}_\rho(\tau \to 0) &\Longrightarrow \forall[][r](\mathsf{M}_r(\tau)) \to 0 \text{ at } \mathsf{cd}
\end{aligned}$$

This definition of M forces the mutator to maintain the invariant that all objects are allocated in the same region, which is all our garbage collector requires.

4.4 Translating λ_{CLOS} to λ_{GC}

The translation of terms from λ_{CLOS} to λ_{GC} shown in Fig. 4.3 is mostly directed by the type translation M_ρ presented earlier: each function takes the current region as an argument and begins by checking if a garbage collection is necessary. All operations on data are slightly

$\boxed{F \vdash \lambda_{\mathsf{CLOS}} \Rightarrow \lambda_{\mathsf{GC}}}$

$$\frac{}{F \vdash_v n \Rightarrow n} \qquad \frac{}{F \vdash_v f \Rightarrow \mathsf{cd}.F(f)} \qquad \frac{}{F \vdash_v x \Rightarrow x}$$

$$\frac{F \vdash_v v_1 \Rightarrow v_1' \quad F \vdash_v v_2 \Rightarrow v_2'}{F \vdash_v \langle v_1, v_2\rangle \Rightarrow \mathsf{put}[r]\langle v_1', v_2'\rangle}$$

$$\frac{F \vdash_v v \Rightarrow v'}{F \vdash_v \langle t{=}\tau_1, v{:}\tau_2\rangle \Rightarrow \mathsf{put}[r]\langle t{=}\tau_1, v'{:}\mathsf{M}_r(\tau_2)\rangle}$$

$$\frac{F \vdash_v v_1 \Rightarrow v_1' \quad F \vdash_v v_2 \Rightarrow v_2'}{F \vdash_e v_1(v_2) \Rightarrow v_1'[\,][r](v_2')} \qquad \frac{F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{halt}\ v \Rightarrow \mathsf{halt}\ v'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{open}\ v\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e \Rightarrow \mathsf{open}\ (\mathsf{get}\ v')\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{let}\ x = v\ \mathsf{in}\ e \Rightarrow \mathsf{let}\ x = v'\ \mathsf{in}\ e'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{let}\ x = \pi_i v\ \mathsf{in}\ e \Rightarrow \mathsf{let}\ x = \pi_i(\mathsf{get}\ v')\ \mathsf{in}\ e'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad \ell = F(f)}{\begin{array}{l}\vdash_f (f = \lambda(x{:}\tau).e) \\ \quad \Rightarrow\ \lambda[\,][r](x{:}\mathsf{M}_r(\tau)).\mathsf{ifgc}\ r\ (gc[\tau][r](\mathsf{cd}.\ell, x))\ e'\end{array}}$$

$$\frac{\begin{array}{c}F = \{f_1 \mapsto \ell_1, \ldots, f_n \mapsto \ell_n\} \\ F \vdash_f f_i = \lambda(x_i{:}\tau_i).e_i \Rightarrow f_i' \qquad F \vdash_e e \Rightarrow e'\end{array}}{\begin{array}{l}\vdash_p \mathsf{letrec}\ \overrightarrow{f_i = \lambda(x_i{:}\tau_i).e_i}\ \mathsf{in}\ e \\ \quad \Rightarrow\ (\{\mathsf{cd} \mapsto \{\ell_1 \mapsto f_1', \ldots\}\}, \mathsf{let\ region}\ r\ \mathsf{in}\ e')\end{array}}$$

Figure 4.3: Translation of λ_{CLOS} terms.

---

rewritten to account for the need to allocate them in the region or to fetch them from the region. For example a λ_{CLOS} function like:

$$\begin{aligned}&\mathsf{fix}\ swap(x\!:\!\mathsf{Int}\times\mathsf{Int}).\\&\quad \mathsf{let}\ x1=\pi_1 x\ \mathsf{in}\ \mathsf{let}\ x2=\pi_2 x\ \mathsf{in}\ \mathsf{let}\ x'=\langle x2,x1\rangle\ \mathsf{in}\ \mathsf{halt}\ 0\end{aligned}$$

will be turned into the following λ_{GC} function:

$$\begin{aligned}&\mathsf{cd}.\ell=\lambda[][r](x\!:\!(\mathsf{int}\times\mathsf{int})\ \mathsf{at}\ r).\\&\qquad \mathsf{ifgc}\ r\ \ (gc[\mathsf{Int}\times\mathsf{Int}][r](\mathsf{cd}.\ell,x))\\&\qquad\quad \mathsf{let}\ x=\mathsf{get}\ x\ \mathsf{in}\\&\qquad\quad \mathsf{let}\ x1=\pi_1 x\ \mathsf{in}\\&\qquad\quad \mathsf{let}\ x2=\pi_2 x\ \mathsf{in}\\&\qquad\quad \mathsf{let}\ x'=\mathsf{put}[r]\langle x2,x1\rangle\ \mathsf{in}\\&\qquad\quad \mathsf{halt}\ 0\end{aligned}$$

The mapping between λ_{CLOS} identifiers like $swap$ and λ_{GC} location like $\mathsf{cd}.\ell$ is kept in F. The new argument r refers to the current region. It is initially created at the very beginning of the program and is changed after each garbage collection.

An important detail here is that the garbage collector receives the tag τ rather than the type σ of the argument. The GC receives the tags for analysis as they were in λ_{CLOS} rather than as they are translated in λ_{GC}. This maintains a clear distinction between the types the programmer thinks he manipulates and the underlying types they map to.

Another interesting detail is that if the region is full, the function calls the garbage collector with itself as the return function. I.e. when the collection is finished, the collector will jump back to the function which will then redo the check. We could instead call the garbage collector with another function as argument. That would save us from redoing the ifgc but would require many tiny functions which are just not worth bothering with.

The translation in Fig. 4.3 uses λ_{GC} in a somewhat loose way to keep the presentation concise. More specifically, it will generate terms such as $\mathsf{let}\ x=\pi_i(\mathsf{get}\ v)\ \mathsf{in}\ e$ instead of

$$\mathsf{let}\ x'=\mathsf{get}\ v\ \mathsf{in}\ \mathsf{let}\ x=\pi_i x'\ \mathsf{in}\ e.$$

$$
\begin{array}{l}
\mathsf{fix}\ gc[t\!:\!\Omega][r_1](f\!:\!\forall[\,][r](\mathsf{M}_r(t)) \to 0, x\!:\!\mathsf{M}_{r_1}(t)).\\
\quad \mathsf{let\ region}\ r_2\ \mathsf{in}\\
\quad \mathsf{let}\ y = \mathit{copy}[t][r_1,r_2](x)\ \mathsf{in}\\
\quad \mathsf{only}\ \{r_2\}\ \mathsf{in}\ f[\,][r_2](y)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fix}\ \mathit{copy}[t\!:\!\Omega][r_1,r_2](x\!:\!\mathsf{M}_{r_1}(t)) : \mathsf{M}_{r_2}(t).\\
\quad \mathsf{tcase}\ t\ \mathsf{of}\\
\qquad \begin{array}{rcl}
\mathsf{Int} & \Rightarrow & x\\
\to & \Rightarrow & x\\
t_1 \times^r t_2 & \Rightarrow & \mathsf{let}\ x_1 = \mathit{copy}[t_1][r_1,r_2](\pi_1(\mathsf{get}\ x))\ \mathsf{in}\\
& & \mathsf{let}\ x_2 = \mathit{copy}[t_2][r_1,r_2](\pi_2(\mathsf{get}\ x))\ \mathsf{in}\\
& & \mathsf{put}[r_2]\langle x_1, x_2\rangle^{r_2}\\
\exists t_e & \Rightarrow & \mathsf{open}\ (\mathsf{get}\ x)\ \mathsf{as}\ \langle t, y\rangle\ \mathsf{in}\\
& & \mathsf{let}\ z = \mathit{copy}[t_e t][r_1,r_2](y)\ \mathsf{in}\\
& & \mathsf{put}[r_2]\langle t\!=\!t, z\!:\!\mathsf{M}_{r_2}(t_e t)\rangle
\end{array}
\end{array}
$$

Figure 4.4: The garbage collector proper.

Turning such code back into the strict λ_{GC} is very straightforward.

On the other hand, the garbage-collection code in Fig. 4.4 uses not only some syntactic sugar but even resorts to using a direct-style presentation of the *copy* function. This is only for clarity of presentation, of course. As can be seen in figure 4.5, the code after CPS and closure conversion is a lot more difficult to read, partly because of the need to do a form of typed closure conversion [MMH96].

The garbage collector itself is very simple: it first allocates the *to* region, asks *copy* to move everything into it and then frees the *from* region before jumping to its continuation, using the new region.

The *copy* function is similarly straightforward, recursing over the whole heap and copying in a depth-first way. Clearly, the direct style here hides the stack. When the code is CPS converted and closed (as shown in Fig. 4.5) , we have to allocate that stack of continuations in an additional temporary region and unless our language is extended with some notion of stack, none of those continuations would be collected until the end of the whole garbage collection. The size of this temporary region can be bounded by the size of the *to* region since we can't allocate more than one continuation per copied object, so it is still algorithmically efficient, although this memory overhead is a considerable shortcoming.

4.4.1 Closed CPS garbage collector

Figure 4.5 presents the code of the basic collector after CPS and closure conversion (the direct-style code is shown in Fig. 4.4). The presence of free tag variables in the continuations requires the use of a form of translucent types for the typed closure conversion [MMH96].

In the general case typed closure conversion also requires existential quantification over kinds, but in the present case, we can avoid it by using a superset of all possible kinds: since some continuations require t_1, t_2 of kind Ω, Ω while others only need t_1, t_e or kind $\Omega, (\Omega \to \Omega)$, we unify the two into t_1, t_2, t_e where some of the arguments are simply left unused.

The verbose type annotations make it look more scary than it really is. Four new functions were introduced because of CPS conversion. They are all used as continuations to calls to *copy*:

1. *gcend* is the code executed at the end of the toplevel call to *copy* and just finishes the garbage collection by freeing the from space and calling back the mutator.

2. *copyexist1* is the continuation of the recursive call to copy an existential package.

3. Since copying a pair requires two recursive calls, we need both *copypair1* for the continuation of the first call and *copypair2* for the continuation of the second.

Since the code has to be closed, all the free variables need to be passed explicitly via the continuation object k whose type is abbreviated as $t_k[t]$ but is really a big existential wrapper around the real data whose type is abbreviated as $t_c[t]$.

4.5 A closer look at λ_{GC}

Programs in λ_{GC} use an allocation semantics which makes the allocation of data in memory explicit. The semantics, defined in Fig. 4.6, maps a machine state P to a new machine

— Syntactic type shorthands for notational convenience: —
— Basic type of the continuations of *copy* —
$t_c[t] \equiv \forall[\![t_1, t_2, t_e]\!][r_1, r_2, r_3](\mathsf{M}_{r_2}(t), \alpha_c) \xrightarrow{\rho} 0 \times \alpha_c$
— Same as t_c but closed with existential packages —
$t_k[t] \equiv (\exists t_1 : \Omega . \exists t_2 : \Omega . \exists t_e : \Omega \to \Omega . \exists \alpha_c : \{r_1, r_2, r_3\} . t_c[t])$ at r_3

— The main *GC* entry point —
$\mathsf{fix}\ gc[t:\Omega][r_1](f : \forall[][r](\mathsf{M}_r(t)) \to 0, x : \mathsf{M}_{r_1}(t)).$
$\quad \mathsf{let\ region}\ r_2\ \mathsf{in}$
$\quad \mathsf{let\ region}\ r_3\ \mathsf{in}$
$\quad \mathsf{let}\ c = \langle gcend[\![t, \mathsf{int}, \lambda t.t]\!], f\rangle\ \mathsf{in}$
$\quad \mathsf{let}\ k = \mathsf{put}[r_3]\langle t_1 = t, t_2 = \mathsf{int}, t_e = \lambda t.t, \alpha_c : \{r_1, r_2, r_3\} = \forall[][r](\mathsf{M}_r(t)) \to 0, c : t_c[t]\rangle\ \mathsf{in}$
$\quad copy[t][r_1, r_2, r_3](x, k)$

— The second half of *GC*, passed as a continuation to *copy* —
$\mathsf{fix}\ gcend[t_1 : \Omega, t_2 : \Omega, t_e : \Omega \to \Omega][r_1, r_2, r_3](y : \mathsf{M}_{r_2}(t_1), f : \forall[][r](\mathsf{M}_r(t_1)) \to 0).$
$\quad \mathsf{only}\ \{r_2\}\ \mathsf{in}\ f[][r_2](y)$

— The main *copy* entry point —
$\mathsf{fix}\ copy[t:\Omega][r_1, r_2, r_3](x : \mathsf{M}_{r_1}(t), k : t_k[t]).$
$\quad \mathsf{tcase}\ t\ \mathsf{of}$
$\qquad \mathsf{Int} \Rightarrow \mathsf{open}\ (\mathsf{get}\ k)\ \mathsf{as}\ \langle t_1, t_2, t_e, \alpha_c, c\rangle\ \mathsf{in}\ (\pi_1 c)[t_1, t_2, t_e][r_1, r_2, r_3](x, \pi_2 c)$
$\qquad \to\ \Rightarrow \mathsf{open}\ (\mathsf{get}\ k)\ \mathsf{as}\ \langle t_1, t_2, t_e, \alpha_c, c\rangle\ \mathsf{in}\ (\pi_1 c)[t_1, t_2, t_e][r_1, r_2, r_3](x, \pi_2 c)$
$\qquad t_1 \times^r t_2 \Rightarrow \mathsf{let}\ c = \langle copypair1[\![t_1, t_2, \lambda t.t]\!], \langle \pi_2(\mathsf{get}\ x), k\rangle\rangle\ \mathsf{in}$
$\qquad\qquad \mathsf{let}\ k = \mathsf{put}[r_3]\langle t_1 = t_1, t_2 = t_2, t_e = \lambda t.t, \alpha_c : \{r_1, r_2, r_3\} = \mathsf{M}_{r_1}(t_2) \times t_k[t], c : t_c[t_1]\rangle$
$\qquad\qquad \mathsf{in}\ copy[t_1][r_1, r_2, r_3](\pi_1(\mathsf{get}\ x), k)$
$\qquad \exists t_e \Rightarrow \mathsf{open}\ (\mathsf{get}\ x)\ \mathsf{as}\ \langle t_x, y\rangle\ \mathsf{in}$
$\qquad\qquad \mathsf{let}\ c = \langle copyexist1[\![t_x, \mathsf{int}, t_e]\!], k\rangle\ \mathsf{in}$
$\qquad\qquad \mathsf{let}\ k = \mathsf{put}[r_3]\langle t_1 = t_x, t_2 = \mathsf{int}, t_e = t_e, \alpha_c : \{r_1, r_2, r_3\} = t_k[t], c : t_c[t_e t_1]\rangle\ \mathsf{in}$
$\qquad\qquad copy[t_e t_x][r_1, r_2, r_3](y, k)$

— First continuation when copying a pair —
$\mathsf{fix}\ copypair1[t_1 : \Omega, t_2 : \Omega, t_e : \Omega \to \Omega][r_1, r_2, r_3](x_1 : \mathsf{M}_{r_2}(t_1), c : \mathsf{M}_{r_1}(t_2) \times t_k[t_1 \times t_2]).$
$\quad \mathsf{let}\ c' = \langle copypair2[\![t_2, t_1, \lambda t.t]\!], \langle x_1, \pi_2 c\rangle\rangle\ \mathsf{in}$
$\quad \mathsf{let}\ k = \mathsf{put}[r_3]\langle t_1 = t_1, t_2 = t_2, t_e = \lambda t.t, \alpha_c : \{r_1, r_2, r_3\} = \mathsf{M}_{r_2}(t_1) \times t_k[t_1 \times t_2], c' : t_c[t_2]\rangle\ \mathsf{in}$
$\quad copy[t_2][r_1, r_2, r_3](\pi_1 c, k)$

— Second continuation when copying a pair —
$\mathsf{fix}\ copypair2[t_1 : \Omega, t_2 : \Omega, t_e : \Omega \to \Omega][r_1, r_2, r_3](x_2 : \mathsf{M}_{r_2}(t_2), c : \mathsf{M}_{r_2}(t_1) \times t_k[t_1 \times t_2]).$
$\quad \mathsf{open}\ (\mathsf{get}\ (\pi_2 c))\ \mathsf{as}\ \langle t_1, t_2, t_e, \alpha_c, c'\rangle\ \mathsf{in}\ (\pi_1 c')[t_1, t_2, t_e][r_1, r_2, r_3](\mathsf{put}[r_2]\langle \pi_1 c, x_2\rangle, \pi_2 c')$

— Continuation when copying an existential package —
$\mathsf{fix}\ copyexist1[t_1 : \Omega, t_2 : \Omega, t_e : \Omega \to \Omega][r_1, r_2, r_3](z : \mathsf{M}_{r_2}(t_e t_1), c : t_k[t_e t_1]).$
$\quad \mathsf{open}\ (\mathsf{get}\ c)\ \mathsf{as}\ \langle t'_1, t'_2, t'_e, \alpha'_c, c'\rangle\ \mathsf{in}\ (\pi_1 c')[t'_1, t'_2, t'_e][r_1, r_2, r_3](\mathsf{put}[r_2]\langle t = t_1, z : \mathsf{M}_{r_2}(t_e t)\rangle, \pi_2 c')$

Figure 4.5: The basic GC code after CPS and closure conversion.

$$
\begin{array}{lll}
(M, \nu.\ell[\vec{\tau}][\vec{\rho}](\vec{v})) & \Longrightarrow & (M, \nu.\ell[\vec{\tau'}][\vec{\rho}](\vec{v})) \\
(M, \nu.\ell[\vec{\tau'}][\vec{\rho}](\vec{v})) & & \\
\quad \text{where } M(\nu.\ell) = (\lambda[t\vec{:}\kappa][\vec{r}](\vec{x}:\vec{\sigma}).e) & \Longrightarrow & (M, e[\vec{\rho}, \vec{\tau'}, \vec{v}/\vec{r}, \vec{t}, \vec{x}]) \\
(M, (v[\![\vec{\tau}]\!])[\vec{\tau}][\vec{\rho}](\vec{v})) & \Longrightarrow & (M, v[\vec{\tau}][\vec{\rho}](\vec{v})) \\
(M, \mathsf{let}\ x = v\ \mathsf{in}\ e) & \Longrightarrow & (M, e[v/x]) \\
(M, \mathsf{let}\ x = \pi_i\langle v_1, v_2\rangle\ \mathsf{in}\ e) & \Longrightarrow & (M, e[v_i/x]) \\
(M, \mathsf{let}\ x = \mathsf{put}[\nu]v\ \mathsf{in}\ e) & \Longrightarrow & (M\{\nu.\ell \mapsto v\}, e[{}^{\nu.\ell}/_{x}]) \quad \text{where } \ell \notin Dom(M(\nu)) \\
(M, \mathsf{let}\ x = \mathsf{get}\ \nu.\ell\ \mathsf{in}\ e) & \Longrightarrow & (M, e[{}^{v}/_{x}]) \quad \text{where } M(\nu.\ell) = v \\
(M, \mathsf{open}\ \langle t{=}\tau, v{:}\sigma\rangle\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e) & \Longrightarrow & (M, \mathsf{open}\ \langle t{=}\tau', v{:}\sigma\rangle\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e) \\
(M, \mathsf{open}\ \langle t{=}\tau', v{:}\sigma\rangle\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e) & \Longrightarrow & (M, e[\tau', v/t, x]) \\
(M, \mathsf{open}\ \langle \alpha{:}\Theta{=}\sigma_1, v{:}\sigma_2\rangle\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ e) & \Longrightarrow & (M, e[\sigma_1, v/\alpha, x]) \\
(M, \mathsf{ifgc}\ \rho\ e_1\ e_2) & \Longrightarrow & (M, e_1) \quad \text{if } \rho \text{ is full} \\
(M, \mathsf{ifgc}\ \rho\ e_1\ e_2) & \Longrightarrow & (M, e_2) \quad \text{if } \rho \text{ is not full} \\
(M, \mathsf{let\ region}\ r\ \mathsf{in}\ e) & \Longrightarrow & (M\{\nu \mapsto \{\}\}, e[\nu/r]) \quad \text{where } \nu \notin Dom(M) \\
(M, \mathsf{only}\ \Theta\ \mathsf{in}\ e) & \Longrightarrow & (M|_{\Theta}, e) \\
(M, \mathsf{tcase}\ \tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists})) & \Longrightarrow & (M, \mathsf{tcase}\ \tau'\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists})) \\
(M, \mathsf{tcase\ Int}\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists})) & \Longrightarrow & (M, e_i) \\
(M, \mathsf{tcase}\ \tau \rightarrow 0\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists})) & \Longrightarrow & (M, e_{\rightarrow}) \\
(M, \mathsf{tcase}\ \tau_1 \times \tau_2\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists})) & \Longrightarrow & (M, e_{\times}[{}^{\tau_1, \tau_2}/_{t_1, t_2}]) \\
(M, \mathsf{tcase}\ \exists t.\tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists})) & \Longrightarrow & (M, e_{\exists}[{}^{\lambda t.\tau}/_{t_e}])
\end{array}
$$

Figure 4.6: Operational semantics of λ_{GC}.

state P'. A machine state is a pair (M, e) of a memory M and a term e being executed. A memory consists of a set of regions; hence, it is defined formally as a map between region names ν and regions R. A region, in turn, is a map from offsets ℓ to storable values v. Therefore, an address is given by a pair of a region and an offset $\nu.\ell$. We assign a type to every location allocated in a region with the memory environment Ψ. Fig. 4.8 presents the static semantics.

4.5.1 Closure conversion and *copy*.

Since the source language is monomorphic, closure conversion need only rely on existentials. This simplicity is however broken by the *copy* function in the garbage collector itself because this function is (recursively) polymorphic. For that reason, we also need a form of translucent type, namely $\forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0$. Closure conversion of the CPS form of *copy* was also the only reason for introducing $\langle \alpha : \Theta = \sigma_1, v : \sigma_2 \rangle$.

4.5.2 Functions and code

Since function bodies can contain references to other functions in cd but we do not have an easy way for the garbage collector to analyze a function body to trace through those references, cd enjoys a special status. It cannot be freed and can only contain functions, no other kind of data.

An alternative would be to require all functions to be fully closed, but that would require the addition of recursive types for the environment containing pointers to all functions and passed around everywhere. It would save us from so many cd special cases, and would allow garbage collecting code, but on the other hand, it would be less realistic since it would amount to disallowing direct function calls.

4.5.3 The type calculus

The target language must be expressive enough to write a tracing garbage collector. Since the garbage collector needs to know the type of values at runtime, the language λ_{GC} must support the runtime analysis of types. Therefore, conceptually, types need to play a dual role in this language. As in the source language λ_{CLOS}, they are used at compile time to type-check well formed terms. However, they are also used at runtime, as tags, to be inspected by the garbage collector (and, in general, by any type analyzing function). To enforce this distinction, we split types into a tag language and a type language. The tags correspond to the runtime entity, while the types correspond to the compile time entity.

During the translation from λ_{CLOS} to λ_{GC}, the tag for a value must be constructed from its type, so the tags in λ_{GC} closely resemble the type language in λ_{CLOS}. To support tag analysis, we need to add tag-level functions ($\lambda t.\tau$) and tag-level applications ($\tau\tau_1$) which in turn requires adding the function kind $\Omega \to \Omega$.

Types are used to classify terms. The type language includes the existential type for typing closures and the code type $\forall[\vec{t}][\vec{r}](\vec{\sigma}) \to 0$ for fully closed CPS functions. Moreover, types in the target language must include the region in which the corresponding value resides. Therefore, we use the notation σ at ρ for the type of a value of type σ in region ρ.

To reason about the safety of programs in this language, we will often need to assume that a value resides in a particular region only. For example, after the copy function is finished, we must be able to assume that all the data is contained only in the new region; so that the old region can be safely freed. Therefore, to ensure type safety, we must be able to enforce this invariant at the type level. For this, we use the built-in type operator M. The type $\mathsf{M}_\rho(\tau)$ can only contain values that are in region ρ. This should be contrasted with values of type σ at ρ which can contain references to other regions than ρ.

4.5.4 The term calculus

The term language must support region based memory management and runtime type analysis. New regions are created through the $\mathsf{let\ region}$ r in e construct which allocates a new region ν at runtime and binds r to it. A term of the form $\mathsf{put}[\rho]v$ allocates a value v in the region ρ. Data is read from a region in two ways. Functions are read implicitly through a function call. Data may also be read through the get v construct.

Operationally, get takes a memory address $\nu.\ell$ and dereferences it. Since our region calculus does not admit dangling references, and since each reference implicitly carries a region handle, get does not need a region argument, as opposed to put.

Deallocation is handled implicitly through the only Θ in e construct [WA99]. It asserts statically that the expression e can be evaluated using only the set of regions in Θ' (i.e. Θ extended with the cd region), which is a subset of the regions currently in scope. At runtime, an implementation would treat the set of regions in Θ' as live and reclaim all other regions.

$$\frac{\Theta' = \Theta, \mathsf{cd} \quad \Psi|_{\Theta'}; \Theta'; \Delta; \Gamma|_{\Theta'} \vdash e \quad \Theta' \subset \Theta''}{\Psi; \Theta''; \Delta; \Gamma \vdash \mathsf{only}\ \Theta\ \mathsf{in}\ e}$$

The construct $|_{\Theta'}$ restricts an environment to the set of regions in Θ'. I.e. $\Psi|_{\Theta'}$ is the subset of the heap restricted to the regions in Θ'. Similarly, $\Gamma|_{\Theta'}$ eliminates from Γ all variables whose type refers to regions not mentioned in Θ'.

The use of only was chosen for its simplicity. Other approaches either do not work with a CPS language or carry a significant added complexity to handle the problem of aliasing whereby the program requests the deletion of region r_1 while r_2 is still in use and might refer to the same region. only side steps this difficulty by making the actual deletion implicit: instead of requesting deletion of r_1, the program will request to keep r_2 which will prevent r_1 from being deleted if r_1 and r_2 happen to be aliases.

The runtime type analysis is handled through a tcase construct (it should arguably be called tagcase since it analyses tags rather than types). Depending on the head of the tag being analyzed, tcase chooses one of the branches for execution. When analyzing a tag variable t, we refine types containing t in each of the branches [CWM98].

$$\frac{\begin{array}{l}\Delta \vdash t : \Omega \\ \Psi; \Theta; \Delta; \Phi; \Gamma[^{\mathsf{int}}/_{t}] \vdash e_i[^{\mathsf{int}}/_{t}] \\ \ldots\end{array}}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \mathsf{tcase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists})}$$

In the e_i branch, we know that the tag variable t is bound to Int and can therefore substitute it away. A similar rule is applied to the other cases.

4.5.5 Formal properties of the language

In this section, we show that type checking in λ_{GC} is decidable and that the calculus is sound. The proofs are in Sec. A.1.

Type checking is decidable

Proposition 4.5.1 *Reduction of well formed types is strongly normalizing.*

Proof Since the tag language is a simply typed lambda calculus, reduction of well formed tags is strongly normalizing and confluent. The termination of $\mathsf{M}_\rho(\tau)$ follows from a straight-

$\boxed{\Delta \vdash \tau : \kappa \quad \Big| \quad \tau \text{ has kind } \kappa \text{ in context } \Delta}$

$$\frac{}{\cdot \vdash \mathsf{Int} : \Omega} \qquad \frac{\Delta(t) = \kappa}{\Delta \vdash t : \kappa} \qquad \frac{\Delta \vdash \tau_1 : \Omega \quad \Delta \vdash \tau_2 : \Omega}{\Delta \vdash \tau_1 \times \tau_2 : \Omega} \qquad \frac{\Delta \vdash \tau_i : \Omega}{\Delta \vdash \vec{\tau} \to 0 : \Omega}$$

$$\frac{\Delta, t{:}\Omega \vdash \tau : \Omega}{\Delta \vdash \exists t.\tau : \Omega} \qquad \frac{\Delta, t{:}\Omega \vdash \tau : \Omega}{\Delta \vdash \lambda t.\tau : \Omega \to \Omega} \qquad \frac{\Delta \vdash \tau_1 : \Omega \to \Omega \qquad \Delta \vdash \tau_2 : \Omega}{\Delta \vdash \tau_1 \tau_2 : \Omega}$$

$\boxed{\Theta; \Delta; \Phi \vdash \sigma \quad \Big| \quad \sigma \text{ is well-formed}}$

$$\frac{}{\Theta; \Delta; \Phi \vdash \mathsf{int}} \qquad \frac{\Theta; \Delta; \Phi \vdash \sigma_1 \qquad \Theta; \Delta; \Phi \vdash \sigma_2}{\Theta; \Delta; \Phi \vdash \sigma_1 \times \sigma_2} \qquad \frac{\{\vec{r}\}; \vec{t{:}\kappa}; \cdot \vdash \sigma_i}{\Theta; \Delta; \Phi \vdash \forall[\vec{t{:}\kappa}][\vec{r}](\vec{\sigma}) \to 0}$$

$$\frac{\Theta; \Delta, t{:}\kappa; \Phi \vdash \sigma}{\Theta; \Delta; \Phi \vdash \exists t{:}\kappa.\sigma} \qquad \frac{\Theta; \Delta; \Phi \vdash \sigma \qquad \rho \in \Theta}{\Theta; \Delta; \Phi \vdash \sigma \ \mathsf{at}\ \rho} \qquad \frac{\Theta; \Delta; \Phi, \alpha{:}\Theta' \vdash \sigma \qquad \Theta' \subset \Theta}{\Theta; \Delta; \Phi \vdash \exists \alpha{:}\Theta'.\sigma}$$

$$\frac{\Delta \vdash \tau : \Omega \qquad \rho \in \Theta}{\Theta; \Delta; \Phi \vdash \mathsf{M}_\rho(\tau)} \qquad \frac{\Phi(\alpha) = \Theta' \qquad \Theta' \subset \Theta}{\Theta; \Delta; \Phi \vdash \alpha} \qquad \frac{\{\vec{r}\}; \Delta; \cdot \vdash \sigma_i \quad \Delta \vdash \tau_i : \kappa_i \quad \rho \in \Theta}{\Theta; \Delta; \Phi \vdash \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0}$$

$\boxed{\begin{array}{l|l} \Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma & v \text{ has type } \sigma \\ \Psi; \Theta; \Delta; \Phi; \Gamma \vdash op : \sigma & op \text{ returns a value of type } \sigma \end{array}}$

$$\frac{}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash n : \mathsf{int}} \qquad \frac{\Gamma(x) = \sigma}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash x : \sigma} \qquad \frac{\Psi(\nu.\ell) = \sigma \quad Dom(\Psi); \cdot; \cdot \vdash \sigma \ \mathsf{at}\ \nu}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \nu.\ell : \sigma \ \mathsf{at}\ \nu}$$

$$\frac{\mathsf{cd}, \vec{r}; \vec{t}; \cdot \vdash \sigma_i \qquad \Psi|_{\mathsf{cd}}; \mathsf{cd}, \vec{r}; \overrightarrow{t{:}\kappa}; \cdot; \overrightarrow{x{:}\sigma} \vdash e}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \lambda[\vec{t{:}\kappa}][\vec{r}](\overrightarrow{x{:}\sigma}).e : \forall[\vec{t{:}\kappa}][\vec{r}](\vec{\sigma}) \to 0}$$

$$\frac{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \forall[\vec{t{:}\kappa}][\vec{r}](\vec{\sigma}) \to 0 \ \mathsf{at}\ \rho \qquad \Delta \vdash \tau_i : \kappa_i}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v[\![\vec{\tau}]\!] : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}[\vec{\tau}/\vec{t}]) \xrightarrow{\rho} 0}$$

$$\frac{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v_1 : \sigma_1 \qquad \Psi; \Theta; \Delta; \Phi; \Gamma \vdash v_2 : \sigma_2}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \langle v_1, v_2 \rangle : \sigma_1 \times \sigma_2} \qquad \frac{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma_1 \times \sigma_2}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \pi_i v : \sigma_i}$$

$$\frac{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma \ \mathsf{at}\ \rho}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \mathsf{get}\ v : \sigma} \qquad \frac{\Delta \vdash \tau : \kappa \qquad \Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma[\tau/t]}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \langle t{=}\tau, v{:}\sigma \rangle : \exists t{:}\kappa.\sigma}$$

$$\frac{\Theta'; \Delta; \Phi|_{\Theta'} \vdash \sigma_1 \qquad \Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma_2[\sigma_1/\alpha]}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \langle \alpha{:}\Theta'{=}\sigma_1, v{:}\sigma_2 \rangle : \exists \alpha{:}\Theta'.\sigma_2} \qquad \frac{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma \qquad \rho \in \Theta}{\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \mathsf{put}[\rho]v : \sigma \ \mathsf{at}\ \rho}$$

Figure 4.7: Static semantics of λ_{GC}, part 1.

$$\boxed{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e \quad \Big| \quad e \text{ is well-formed}}$$

$$\frac{\begin{array}{c}\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \forall[\vec{t\!:\!\kappa}][\vec{r}](\vec{\sigma}) \to 0 \text{ at } \rho \\ \Psi;\Theta;\Delta;\Phi;\Gamma \vdash v_i : \sigma_i[{}^{\vec{\rho},\vec{\tau}}/_{\vec{r},\vec{t}}] \qquad \Delta \vdash \tau_i : \kappa_i \qquad \rho_i \in \Theta\end{array}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v[\vec{\tau}][\vec{\rho}](\vec{v})}$$

$$\frac{\begin{array}{c}\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0 \\ \Psi;\Theta;\Delta;\Phi;\Gamma \vdash v_i : \sigma_i[{}^{\vec{\rho}}/_{\vec{r}}] \qquad \rho_i \in \Theta\end{array}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v[\vec{\tau}][\vec{\rho}](\vec{v})} \qquad \frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash op : \sigma \qquad \Psi;\Theta;\Delta;\Phi;\Gamma, x\!:\!\sigma \vdash e}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{let}\ x = op\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \exists t'\!:\!\kappa.\sigma \quad \Psi;\Theta;\Delta,t\!:\!\kappa;\Phi;\Gamma,x\!:\!\sigma[{}^{t}/_{t'}] \vdash e}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{open}\ v\ \mathsf{as}\ \langle t, x\rangle\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \exists \alpha'\!:\!\Theta'.\sigma \quad \Psi;\Theta;\Delta;\Phi,\alpha\!:\!\Theta';\Gamma,x\!:\!\sigma[{}^{\alpha}/_{\alpha'}] \vdash e}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{open}\ v\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_1 \quad \Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_2 \quad \rho \in \Theta}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{ifgc}\ \rho\ e_1\ e_2} \qquad \frac{\Psi;\Theta,r;\Delta;\Phi;\Gamma \vdash e}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{let\ region}\ r\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \mathsf{int}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{halt}\ v} \qquad \frac{\Psi|_{\Theta'};\Theta',\mathsf{cd};\Delta;\Phi|_{\Theta'};\Gamma|_{\Theta'} \vdash e \qquad \Theta' \subset \Theta}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{only}\ \Theta'\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_{\mathsf{i}}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{tcase\ Int\ of}\ (e_{\mathsf{i}}; e_{\to}; t_1t_2.e_{\times}; t_e.e_{\exists})}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_{\to}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{tcase}\ \vec{\tau} \to 0\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1t_2.e_{\times}; t_e.e_{\exists})}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_{\times}[{}^{\tau_1,\tau_2}/_{t_1,t_2}]}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{tcase}\ (\tau_1 \times^{\rho} \tau_2)\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1t_2.e_{\times}; t_e.e_{\exists})}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_{\exists}[{}^{\lambda t.\tau}/_{t_e}]}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{tcase}\ \exists t.\tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1t_2.e_{\times}; t_e.e_{\exists})}$$

$$\frac{\begin{array}{l}\Delta \vdash t : \Omega \\ \Psi;\Theta;\Delta;\Phi;\Gamma[{}^{\mathsf{int}}/_{t}] \vdash e_{\mathsf{i}}[{}^{\mathsf{int}}/_{t}] \\ \Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_{\to} \\ \Psi;\Theta;\Delta,t_1\!:\!\Omega,t_2\!:\!\Omega;\Phi;\Gamma[{}^{t_1 \times^r t_2}/_{t}] \vdash e_{\times}[{}^{t_1 \times^r t_2}/_{t}] \\ \Psi;\Theta;\Delta,t_e\!:\!\Omega \to \Omega;\Phi;\Gamma[{}^{\exists t.t_e t}/_{t}] \vdash e_{\exists}[{}^{\exists t.t_e t}/_{t}]\end{array}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{tcase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; t_1t_2.e_{\times}; t_e.e_{\exists})}$$

Figure 4.8: Static semantics of λ_{GC}, part 2.

$\Theta \vdash \Upsilon$	Υ is well-formed in context Θ
$\vdash \Psi$	Ψ is well-formed

$$\frac{\Theta; \cdot; \cdot \vdash \sigma_i}{\Theta \vdash \{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n\}} \qquad \frac{\{\nu_1, \ldots, \nu_n\} \vdash \Upsilon_i}{\vdash \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}}$$

$$\Upsilon_{\mathsf{cd}} = \{\ell_1 : \forall[\vec{\tau_1}][\vec{r_1}](\overrightarrow{v_1 : \sigma_1}) \to 0, \ldots, \ell_n : \forall[\vec{\tau_n}][\vec{r_n}](\overrightarrow{v_n : \sigma_n}) \to 0\}$$

$\Psi \vdash R : \Upsilon$	R has type Υ in context Ψ
$\vdash M : \Psi$	M has type Ψ

$$\frac{\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_i : \sigma_i}{\Psi \vdash \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} : \{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n\}}$$

$$\frac{\vdash \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\} \qquad \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\} \vdash R_i : \Upsilon_i}{\vdash \{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\} : \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}}$$

Figure 4.9: Environment formation rules.

forward induction on the size of the tag τ. □

Proposition 4.5.2 *Reduction of well formed types is confluent.*

Proof Since the reduction of well formed types is strongly normalizing, confluence of the reduction follows from local confluence. This follows easily from a case analysis of the reduction of the $\mathsf{M}_\rho(\tau)$ tag. □

Soundness

Definition 4.5.3 *The judgment* $\vdash (M, e)$ *says that the machine state* (M, e) *is well-formed. It is defined by:*

$$\frac{\vdash M : \Psi \qquad \Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash e}{\vdash (M, e)}$$

Contrary to the other environments, Ψ is not explicitly constructed in any of the static rules, since it reflects dynamic information. Instead, the soundness proof, or more specifically the type preservation proof, needs to construct some witness Ψ' for the new state (M', e') based on the Ψ of the initial state (M, e).

Proposition 4.5.4 (Type Preservation)

If $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.

Proposition 4.5.5 (Progress)

If $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* (M', e') *such that* $(M, e) \Longrightarrow (M', e')$.

4.6 Forwarding pointers

The base algorithm presented before is unrealistic in a number of ways. The first is the fact that the *copy* function does not preserve sharing and thus turns any DAG into a tree.

We hence need to add some form of *forwarding pointers*. Wang and Appel [WA01] propose to pair up every object with its forwarding pointer, incurring a significant memory cost. This additional word is not unheard of, since replicating garbage collectors [NO93, BC99] incur a similar overhead, justified by the desire to provide concurrent collection while avoiding the cost of a read-barrier. We want instead to represent objects as a sum $(\sigma + \mathsf{fwd}\ \sigma)$, which requires a single bit per object and corresponds much more closely to traditional implementations. To this end, $\lambda_{\mathsf{GCforw}}$ extends λ_{GC} with new types and terms for tag bits and sum types as well as memory assignment. We do not need a new fwd or ref type since we can use the region calculus' references for that purpose.

Another requirement for a realistic GC is that the mutator should not need to constantly check for the presence of forwarding pointers since such a read-barrier would only be justified for an incremental GC. In other words, the type as seen by the mutator should not be a sum, although it should still contain the single-bit tag that the GC will use to distinguish between forwarding pointers. Also there should be a way to switch from the mutator's view of the type of an object to the one of the collector. So we also need to add a form of cast that we call widen which we will use at the beginning of a collection to give the collector access to the forwarding pointers:

$\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \sigma$	v has type σ
$\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e$	e is well-formed

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \mathsf{left}\ \sigma}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{strip}\ v : \sigma} \qquad \frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \mathsf{right}\ \sigma}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{strip}\ v : \sigma}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \sigma}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{inl}\ v : \mathsf{left}\ \sigma} \qquad \frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \sigma}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{inr}\ v : \mathsf{right}\ \sigma}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \mathsf{left}\ \sigma_1 \qquad \Theta;\Delta;\Phi \vdash \mathsf{right}\ \sigma_2}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \mathsf{left}\ \sigma_1 + \mathsf{right}\ \sigma_2}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \mathsf{right}\ \sigma_2 \qquad \Theta;\Delta;\Phi \vdash \mathsf{left}\ \sigma_1}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \mathsf{left}\ \sigma_1 + \mathsf{right}\ \sigma_2}$$

$$\frac{\begin{array}{c}\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \sigma_1 + \sigma_2 \\ \Psi;\Theta;\Delta;\Phi;\Gamma, x{:}\sigma_1 \vdash e_l \quad \Psi;\Theta;\Delta;\Phi;\Gamma, x{:}\sigma_2 \vdash e_r\end{array}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{ifleft}\ x = v\ e_l\ e_r}$$

$$\frac{\begin{array}{c}\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e \\ \Psi;\Theta;\Delta;\Phi;\Gamma \vdash v_1 : \sigma\ \mathsf{at}\ \rho \quad \Psi;\Theta;\Delta;\Phi;\Gamma \vdash v_2 : \sigma\end{array}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{set}\ v_1 := v_2\ ;\ e}$$

$$\frac{\begin{array}{c}\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \mathsf{M}_\rho(\tau) \\ \Psi|_{\mathsf{cd}};\mathsf{cd},\rho,\rho';\Delta;\Phi|_{\rho\rho'};x{:}\mathsf{C}_{\rho,\rho'}(\tau) \vdash e\end{array}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{let}\ x = \mathsf{widen}[\rho'][\tau](v)\ \mathsf{in}\ e}$$

Figure 4.10: Static semantics for $\lambda_{\mathsf{GCforw}}$.

$$\begin{array}{lcl}
\sigma & ::= & \ldots \mid \mathsf{left}\ \sigma \mid \mathsf{right}\ \sigma \mid \mathsf{left}\ \sigma_1 + \mathsf{right}\ \sigma_2 \mid \mathsf{C}_{\rho,\rho'}(\tau) \\
v & ::= & \ldots \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \\
op & ::= & \ldots \mid \mathsf{strip}\ v \\
e & ::= & \ldots \mid \mathsf{ifleft}\ x = v\ e_l\ e_r \mid \mathsf{set}\ v_1 := v_2\ ;\ e \mid \mathsf{let}\ x = \mathsf{widen}[\rho][\tau](v)\ \mathsf{in}\ e
\end{array}$$

Figure 4.10 shows the relevant new rules of the static semantics. Here, inl and inr (and their type-level counterparts left and right) can be thought of as adding a single tag bit to an object while strip gets back the untagged object and ifleft checks the tag bit. The idea is to represent objects as ($\mathsf{left}\ \sigma$) to the mutator (and without the "$\mathsf{right}\ \sigma$" alternative to avoid the need for checks) and cast them with the widen operator to ($\mathsf{left}\ \sigma + \mathsf{right}(\sigma\ \mathsf{at}\ to)$) when entering the garbage collector (here "to" denotes the region for the to-space).

Since a single source-level type now maps to two different possible types, we need two type operators: $\mathsf{M}_\rho(\tau)$ to map source types to the mutator's view of the data and $\mathsf{C}_{\rho,\rho'}(\tau)$ to map source types to the collector's view (which adds forwarding pointers). $\mathsf{M}_\rho(\tau)$ is the same as before for base types and for code types, but is changed for existentials and pairs by adding the left constructor that constrains the mutator to provide the tag bit needed to distinguish the forwarded pointer from the non-forwarded data.

$$
\begin{array}{ll}
\mathsf{M}_\rho(\mathsf{Int}) & \Longrightarrow \mathsf{int} \\
\mathsf{M}_\rho(\tau \to 0) & \Longrightarrow \forall[][r](\mathsf{M}_r(\tau)) \to 0\ \mathsf{at}\ \mathsf{cd} \\
\mathsf{M}_\rho(\exists t.\tau) & \Longrightarrow (\mathsf{left}(\exists t.\mathsf{M}_\rho(\tau)))\ \mathsf{at}\ \rho \\
\mathsf{M}_\rho(\tau_1 \times \tau_2) & \Longrightarrow (\mathsf{left}(\mathsf{M}_\rho(\tau_1) \times \mathsf{M}_\rho(\tau_2)))\ \mathsf{at}\ \rho
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{C}_{\rho,\rho'}(\mathsf{Int}) & \Longrightarrow \mathsf{int} \\
\mathsf{C}_{\rho,\rho'}(\tau \to 0) & \Longrightarrow \mathsf{M}_\rho(\tau \to 0) \\
\mathsf{C}_{\rho,\rho'}(\exists t.\tau) & \Longrightarrow (\mathsf{left}(\exists t.\mathsf{C}_{\rho,\rho'}(\tau)) + \mathsf{right}(\mathsf{M}_{\rho'}(\exists t.\tau)))\ \mathsf{at}\ \rho \\
\mathsf{C}_{\rho,\rho'}(\tau_1 \times \tau_2) & \Longrightarrow (+\begin{array}{l}\mathsf{left}(\mathsf{C}_{\rho,\rho'}(\tau_1) \times \mathsf{C}_{\rho,\rho'}(\tau_2)) \\ \mathsf{right}(\mathsf{M}_{\rho'}(\tau_1 \times \tau_2))\end{array})\ \mathsf{at}\ \rho
\end{array}
$$

It is worth noting again here how the M type operators cleanly encapsulate the invariants imposed on the mutator by the collector. In this case, it forces the mutator to provide the collector with free bit that the collector can then use to distinguish forwarding pointers from non-forwarded data. And we also see how the same mechanism can be used to express the difference between the restricted view offered to the mutator and the full blown access to internal data that the collector needs.

The operational semantics of the new operations is straightforward, especially since we can implement the assignment operator by reusing the indirection through the memory:

$$\begin{aligned}
&(M, \mathsf{let}\ x = \mathsf{strip}\ (\mathsf{inl}\ v)\ \mathsf{in}\ e) &&\Longrightarrow (M, e[v/x])\\
&(M, \mathsf{let}\ x = \mathsf{strip}\ (\mathsf{inr}\ v)\ \mathsf{in}\ e) &&\Longrightarrow (M, e[v/x])\\
&(M, \mathsf{ifleft}\ x = (\mathsf{inl}\ v)\ e_l\ e_r) &&\Longrightarrow (M, e_l[\mathsf{inl}\ v/x])\\
&(M, \mathsf{ifleft}\ x = (\mathsf{inr}\ v)\ e_l\ e_r) &&\Longrightarrow (M, e_r[\mathsf{inr}\ v/x])\\
&(M, \mathsf{set}\ \nu.\ell := v\ ;\ e) &&\Longrightarrow (M\{\nu.\ell \mapsto v\}, e)\\
&(M, \mathsf{let}\ x = \mathsf{widen}[\rho][\tau](v)\ \mathsf{in}\ e) &&\Longrightarrow (M, e[v/x])
\end{aligned}$$

The translation from λ_{CLOS} to this $\lambda_{\mathsf{GCforw}}$ is not shown since it is basically the same as before except for the insertion of all the inl and strip. The garbage collector can be seen in Fig. 4.11. Compared to the original algorithm, the only difference in the *gc* function itself is the widening of the heap from M_{r1} to $\mathsf{C}_{r1,r2}$ and the fact that we have to bundle the f and x arguments into a pair in order to pass it through the widen operator and unbundle it afterwards. The *copy* function also needs to be changed of course: when copying a heap object such as a pair, it now has to check with ifleft whether the object was forwarded, if so it just returns the forwarded object, otherwise it does the copy as before and has to overwrite (using set) the original object with the forwarding pointer before returning the copied object. Since the copy is still fundamentally depth-first, it will loop indefinitely in the presence of a cycle. Our source calculus cannot create cycles in the heap, but for more realistic languages, this is a significant restriction.

4.6.1 How to widen safely

The only non-trivial extension is widen which allows the garbage collector to have a different view of the existing memory, provided the two views are somehow *compatible*. It seems difficult to solve the problem of allowing two views on the same data without such a form of cast. At first, it seems we are just applying a form of subtyping, but this form of subtyping is very powerful since it allows covariant subtyping of references. This means that aliasing issues have to be handled with extreme care.

When faced with the same problem, Wang and Appel came up independently with a

$$
\begin{array}{l}
\mathsf{fix}\ gc[t\!:\!\Omega][r_1](f\!:\!\mathsf{M}_{r_1}(t \to 0), x\!:\!\mathsf{M}_{r_1}(t)).\\
\quad \mathsf{let\ region}\ r_2\ \mathsf{in}\\
\quad \mathsf{let}\ w = \mathsf{widen}[r_2][(t \to 0 \times t)](\mathsf{put}[r_1](\mathsf{inl}\ \langle f, x\rangle))\ \mathsf{in}\\
\quad \mathsf{ifleft}\ w = \mathsf{get}\ w\ \mathsf{then}\\
\qquad \mathsf{let}\ w = \mathsf{strip}\ w\ \mathsf{in}\\
\qquad \mathsf{let}\ y = \mathit{copy}[t][r_1, r_2](\pi_2 w)\ \mathsf{in}\\
\qquad \mathsf{only}\ \{r_2\}\ \mathsf{in}\ (\pi_1 w)[][r_2](y)\\
\quad \mathsf{else}\\
\qquad \mathsf{halt}\ 0
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fix}\ \mathit{copy}[t\!:\!\Omega][r_1, r_2](x\!:\!\mathsf{C}_{r_1,r_2}(t)) : \mathsf{M}_{r_2}(t).\\
\quad \mathsf{tcase}\ t\ \mathsf{of}\\
\qquad \begin{array}{rl}
\mathsf{Int} & \Rightarrow x\\
\to & \Rightarrow x\\
t_1 \times^r t_2 & \Rightarrow \mathsf{let}\ y = \mathsf{get}\ x\ \mathsf{in}\\
& \quad\ \mathsf{ifleft}\ y = y\ \mathsf{then}\\
& \qquad \mathsf{let}\ x_1 = \mathit{copy}[t_1][r_1, r_2](\pi_1(\mathsf{strip}\ y))\ \mathsf{in}\\
& \qquad \mathsf{let}\ x_2 = \mathit{copy}[t_2][r_1, r_2](\pi_2(\mathsf{strip}\ y))\ \mathsf{in}\\
& \qquad \mathsf{let}\ z = \mathsf{put}[r_2](\mathsf{inl}\ \langle x_1, x_2\rangle^{r_2})\ \mathsf{in}\\
& \qquad \mathsf{set}\ x := \mathsf{inr}\ z\ ;\ z\\
& \quad\ \mathsf{else}\\
& \qquad \mathsf{strip}\ y\\
\exists t_e & \Rightarrow \mathsf{let}\ y = \mathsf{get}\ x\ \mathsf{in}\\
& \quad\ \mathsf{ifleft}\ y = y\ \mathsf{then}\\
& \qquad \mathsf{open}\ (\mathsf{strip}\ y)\ \mathsf{as}\ \langle t, y\rangle\ \mathsf{in}\\
& \qquad \mathsf{let}\ y = \mathit{copy}[t_e t][r_1, r_2](y)\ \mathsf{in}\\
& \qquad \mathsf{let}\ z = \mathsf{put}[r_2](\mathsf{inl}\ \langle t{=}t, y\!:\!\mathsf{M}_{r_2}(t_e t)\rangle)\ \mathsf{in}\\
& \qquad \mathsf{set}\ x := \mathsf{inr}\ z\ ;\ z\\
& \quad\ \mathsf{else}\\
& \qquad \mathsf{strip}\ y
\end{array}
\end{array}
$$

Figure 4.11: GC with forwarding pointers.

---

similar idea. But their suggested cast leaves many questions open and might need more work to be made type-safe. Also its operational semantics actually does a complete copy of the heap from one region to the other. This might make it easier to prove soundness but makes it unclear whether it can really be implemented as a nop. In contrast, the operational semantics of widen is a nop and we have a proof of its soundness.

In order to handle the problem of aliasing mentioned above, it might be possible to rely on some form of linear typing or alias types [WM00], but given the inherent generality of a garbage collector, it seems difficult. Our approach is to rely on the consistent application of the same cast over the whole heap, so that aliases are guaranteed to be cast in the same way (or discarded, since all free variables are thrown away).

Rather than an ad-hoc widen we could provide a more general cast that consistently applies a given type transformation (as long as it obeys the notion of subtyping extended with covariant subtyping of references) to any particular set of regions. See the next chapter for an example.

In Figure 4.10, the typing rule for widen shows that the expression e is typed in an environment that only contains x. In essence x represents the entire heap. Further, x is obtained from the value v that has type $\mathsf{M}_\rho(\tau)$. Looking at the definition of M, we can see that all values reachable from v will have a type of the form $\mathsf{M}_\rho(\tau')$. Since both M and C are iterators, we can now define a casting operation from one type to the other as an iterator. This iterator will traverse the entire heap and systematically convert from one type to the other; this systematic conversion is necessary to avoid ending up with a value that has a particular type along one path, but has a different type along another path.

The proof of soundness of widen is rather intricate. It starts by ignoring some dead objects from the heap, so that only objects of type $\mathsf{M}_\rho(\tau)$ are left, which get cast to $\mathsf{C}_{\rho,\rho'}(\tau)$. For that reason, we need to loosen our notion of a well formed machine state to allow restricting the considered memory M to just a well-typed sufficient subset $\overline{M}$, where "sufficient" means that no object outside of $\overline{M}$ is needed to complete execution. This safely permits ill-typed garbage.

Definition 4.6.1 *The machine state (M, e) is well formed iff*

$$\frac{\overline{M} \subset M \quad \vdash \overline{M} : \Psi \quad \Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash e}{\vdash (M, e)}$$

Proposition 4.6.2 (Type Preservation)

If $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.

Proof See Sec. A.1.2.

Proposition 4.6.3 (Progress)

If $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* (M', e') *such that* $(M, e) \Longrightarrow (M', e')$.

Proof See Sec. A.1.2.

4.7 Generational collection

Another important aspect of a modern GC is the support for generational garbage collection. If we first restrict ourselves to a side-effect free language, then we can collect a single generation at a time so long as we can express the fact that an object in the old generation cannot point to an object in the young generation.

To that end we need to extend λ_{GC} with existential quantification over regions, so that the mutator does not need to care whether an object is allocated in the young or the old region. We also need to add some way to check in which region an object is allocated so that GC can detect when an object is in the old generation (and hence does not need copying):

$$\begin{aligned}
\sigma &::= \ldots \mid \exists r \in \Theta.(\sigma \text{ at } r) \\
v &::= \ldots \mid \langle r \in \Theta = \rho, v : \sigma \rangle \\
e &::= \ldots \mid \mathsf{open}\ v\ \mathsf{as}\ \langle r, x \rangle\ \mathsf{in}\ e \mid \mathsf{ifreg}\ (\rho_1 = \rho_2)\ e_1\ e_2
\end{aligned}$$

Apart from those new constructs (whose static semantics is presented in Fig. 4.12), the M type operator also needs to be modified to reflect the new invariant imposed on the mutator. It is now indexed by two regions (the old and the new) and has to enforce the fact that objects in the old region cannot have references to the new region:

$$\begin{aligned}
\mathsf{M}_{\rho_y,\rho_o}(\mathsf{Int}) &\Longrightarrow \mathsf{int} \\
\mathsf{M}_{\rho_y,\rho_o}(\tau \to 0) &\Longrightarrow \forall[][r_y, r_o](\mathsf{M}_{r_y,r_o}(\tau)) \to 0 \text{ at cd} \\
\mathsf{M}_{\rho_y,\rho_o}(\exists t.\tau) &\Longrightarrow \exists r \in \{\rho_y, \rho_o\}.((\exists t.\mathsf{M}_{r,\rho_o}(\tau)) \text{ at } r) \\
\mathsf{M}_{\rho_y,\rho_o}(\tau_1 \times \tau_2) &\Longrightarrow \exists r \in \{\rho_y, \rho_o\}.((\mathsf{M}_{r,\rho_o}(\tau_1) \times \mathsf{M}_{r,\rho_o}(\tau_2)) \text{ at } r)
\end{aligned}$$

By using the set $\{r, \rho_o\}$ we make sure that if r is the old generation, pointers reachable

$\Theta;\Delta;\Phi \vdash \sigma$	σ is well-formed
$\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \sigma$	v has type σ
$\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e$	e is well-formed

$$\frac{\Theta' \subset \Theta \qquad \Theta, r;\Delta;\Phi \vdash \sigma}{\Theta;\Delta;\Phi \vdash \exists r \in \Theta'.(\sigma \text{ at } r)} \qquad \frac{\Delta \vdash \tau : \Omega \qquad \rho_1 \in \Theta \qquad \rho_2 \in \Theta}{\Theta;\Delta;\Phi \vdash \mathsf{M}_{\rho_1,\rho_2}(\tau)}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \sigma[^{\rho}/_{r}] \text{ at } \rho \qquad \rho \in \Theta' \qquad \Theta' \subset \Theta}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \langle r \in \Theta' = \rho, v{:}\sigma\rangle : \exists r \in \Theta'.(\sigma \text{ at } r)}$$

$$\frac{\begin{array}{c}\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \exists r \in \Theta'.(\sigma \text{ at } r) \\ \Psi;\Theta, r;\Delta;\Phi;\Gamma, x{:}\sigma \text{ at } r \vdash e\end{array}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{open}\ v \text{ as } \langle r, x\rangle \text{ in } e}$$

$$\frac{\begin{array}{l}\Psi;\Theta[^{r,r}/_{r_1,r_2}];\Delta;\Phi[^{r,r}/_{r_1,r_2}];\Gamma[^{r,r}/_{r_1,r_2}] \vdash e_1[^{r,r}/_{r_1,r_2}] \\ \Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_2 \qquad r \notin \Theta\end{array}}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{ifreg}\ (r_1 = r_2)\ e_1\ e_2}$$

$$\frac{\Psi;\Theta[^{\nu}/_{r}];\Delta;\Phi[^{\nu}/_{r}];\Gamma[^{\nu}/_{r}] \vdash e_1[^{\nu}/_{r}] \qquad \Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_2}{\begin{array}{c}\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{ifreg}\ (r = \nu)\ e_1\ e_2 \\ \Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{ifreg}\ (\nu = r)\ e_1\ e_2\end{array}}$$

$$\frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_2}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{ifreg}\ (\nu_1 = \nu_2)\ e_1\ e_2} \qquad \frac{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_1}{\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \mathsf{ifreg}\ (\nu_1 = \nu_1)\ e_1\ e_2}$$

Figure 4.12: Static semantics of λ_{GCgen}.

from it cannot point back to the new generation.

The operational semantics are again rather simple:

$$\begin{array}{ll}(M, \mathsf{open}\ \langle r \in \Theta = \nu, v{:}\sigma\rangle \text{ as } \langle r, x\rangle \text{ in } e) & \Longrightarrow (M, e[\nu, v/r, x]) \\ (M, \mathsf{ifreg}\ (\nu = \nu)\ e_1\ e_2) & \Longrightarrow (M, e_1) \\ (M, \mathsf{ifreg}\ (\nu_1 = \nu_2)\ e_1\ e_2) & \Longrightarrow (M, e_2)\end{array}$$

Although the operational semantics do not take advantage of it (in order to simplify the soundness proof), we defined the existentials over regions in such a way that they can be implemented as nop since the encapsulated reference usually already encodes the region in its bit-pattern (or in its $\nu.\ell$).

The new term translation is again not shown since it is so similar to the original one. The new type constraint is trivially always satisfied as long as the mutator only allocates from the younger generation and as long as the memory is immutable. If side-effects were to be necessary, it should be possible to extend this scheme with one mutable region (keeping all others immutable) which would be considered similarly to the older generation but scanned at each collection. Obviously, this would first require adding some way to scan a region, but should not present any serious difficulty.

The GC itself can be seen in figure 4.13. The main difference with the basic GC of figure 4.4 is that it does not copy to a new region but to an existing one and stops traversing the tree as soon as we encounter a reference to the old generation.

When hitting such an external reference, we have to repack it just to help the type-system understand that this reference is of type $\mathsf{M}_{\rho_y,\rho_o}(\tau)$. But those operations are free anyway.

Note that another function needs to be written to garbage collect the old generation, but that one is the same as the non-generational one.

At first sight, the λ_{GCgen} language may seem unsound because we allow existentials over regions. However, these types are not existentials in a real sense since they do not hide a region within a type. Rather, in the type $\exists r \in \Theta.(\sigma \text{ at } r)$, the set Θ is an upper bound on the regions that the variable r may range over. In this sense, our existential is closer to a bounded quantification.

Proposition 4.7.1 (Type Preservation)

If $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.

Proof See Sec. A.1.3.

Proposition 4.7.2 (Progress)

If $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* (M', e') *such that* $(M, e) \Longrightarrow (M', e')$.

Proof See Sec. A.1.3.

$$
\begin{array}{l}
\textsf{fix}\ gc[t\!:\!\Omega][r_y,r_o](f\!:\!\mathsf{M}_{r_y,r_o}(t\to 0), x\!:\!\mathsf{M}_{r_y,r_o}(t)).\\
\quad \textsf{let}\ y = copy[t][r_y,r_o](x)\ \textsf{in}\\
\quad \textsf{only}\ \{r_o\}\ \textsf{in let region}\ r_y\ \textsf{in}\ f[][r_y,r_o](y)
\end{array}
$$

$$
\begin{array}{l}
\textsf{fix}\ copy[t\!:\!\Omega][r_y,r_o](x\!:\!\mathsf{M}_{r_y,r_o}(t)) : \mathsf{M}_{r_o,r_o}(t).\\
\quad \textsf{tcase}\ t\ \textsf{of}\\
\qquad \begin{array}{lll}
\textsf{Int} & \Rightarrow & x\\
\to & \Rightarrow & x\\
t_1 \times^r t_2 & \Rightarrow & \textsf{open}\ x\ \textsf{as}\ \langle r,x\rangle\ \textsf{in}\\
& & \textsf{ifreg}\ r = r_o\ \textsf{then}\ \langle r\in\{r_o\}=r_o, x\rangle\ \textsf{else}\\
& & \quad \textsf{let}\ x_1 = copy[t_1][r_y,r_o](\pi_1(\textsf{get}\ x))\ \textsf{in}\\
& & \quad \textsf{let}\ x_2 = copy[t_2][r_y,r_o](\pi_2(\textsf{get}\ x))\ \textsf{in}\\
& & \quad \langle r\in\{r_o\}=r_o, \textsf{put}[r]\langle x_1,x_2\rangle^{r_2}\rangle\\
\exists t_e & \Rightarrow & \textsf{open}\ x\ \textsf{as}\ \langle r,x\rangle\ \textsf{in}\\
& & \textsf{ifreg}\ r = r_o\ \textsf{then}\ \langle r\in\{r_o\}=r_o, x\rangle\ \textsf{else}\\
& & \quad \textsf{open}\ (\textsf{get}\ x)\ \textsf{as}\ \langle t,y\rangle\ \textsf{in}\\
& & \quad \textsf{let}\ z = copy[t_e t][r_y,r_o](y)\ \textsf{in}\\
& & \quad \langle r\in\{r_o\}=r_o, \textsf{put}[r]\langle t\!=\!t, z\!:\!\mathsf{M}_{r,r_o}(t_e t)\rangle\rangle
\end{array}
\end{array}
$$

Figure 4.13: Generational GC.

4.8 Related work

Wang and Appel [WA99] proposed to build a tracing garbage collector on top of a region-based calculus, thus providing both type safety and completely automatic memory management. The main weakness of their proposal is that it relies on a closure conversion algorithm due to Tolmach [TO98] that represents closures as datatypes. This makes closures transparent, making it easier for the copy function to analyze, but it requires whole program analysis and has major drawbacks in the presence of separate compilation. We believe it is more natural to represent closures as existentials [MMH96, MH97] and we show how to use intensional type analysis (on quantified types [TSS00]) to typecheck the GC-copy function.

Recently, Vanderwaart and Crary [VC03] have focused on type-checking the details of the interface between the mutator and the collector, without trying to type-check the GC itself. They show how to express in the types the constraints on the shape of the stack that are necessary for the GC to find all the roots. Also the GC function is not called explicitly, instead the typing rules enforce that every point where a GC can occur has a well-formed stack. Their modeling of the stack layout is very detailed and includes for example a table that maps return addresses to the description of the contents of the corresponding activation frames.

Intensional type analysis was first proposed by Harper and Morrisett [HM95]. They introduced the idea of having explicit type analysis operators which inductively traverse the structure of types. However, to retain decidability of type checking, they restrict the analysis to a predicative subset of the type language. Crary et al. [CW99] propose a very powerful type analysis framework. They define a rich kind calculus that includes sum kinds and inductive kinds. They also provide primitive recursion at the type level. Therefore, they can define new kinds within their calculus and directly encode type analysis operators within their language. They also include a novel refinement operation at the term level. Saha et al [TSS00] shows how to handle polymorphic functions that analyze the quantified type variable—this allows the type analysis to handle arbitrary quantified types. The typerec operators (e.g., M_ρ) used in this paper do not require the full power of what is provided in [TSS00] because our source language is only a simply typed lambda calculus.

Tofte and Talpin [TT94] proposed to use region calculus to type check memory management for higher-order functional languages. Crary et al [CWM99] presented a low-level typed intermediate language that can express explicit region allocation and deallocation. Our λ_{GC} language borrows the basic organization of memories and regions from Crary et al. The main difference is that we don't require explicit capabilities—region deallocation is handled through the only primitive.

Necula and Lee [Nec97, NL98] proposed the idea of proof-carrying code and showed how to construct a certifying compiler for a type-safe subset of C. Morrisett et al. [MWCG98] showed that type-preserving compilation via typed assembly language is a good basis for building certifying compilers. This paper shows that low-level runtime services such as garbage collection can also be expressed in a type-safe language.

4.9 Conclusion

I have presented a type-safe intermediate language with regions and intensional type analysis and show how it can be used to provide a simple and provably type-safe stop-and-copy tracing garbage collector. My key idea is to use intensional type analysis on quantified types (i.e., existentials) to express the garbage-collection invariants on the mutator data objects. I show how this same idea can be used to express more realistic scavengers with efficient forwarding pointers and generations. Because intensional type analysis is also applicable to polymorphic lambda calculus [TSS00], we believe our type safe collector can be extended to handle polymorphic languages as well.

Chapter 5

Typed regions

As the technology of certifying compilation and proof carrying code [Nec97, App01, HST$^+$02] progresses, the need to ensure the safety of the runtime system increases: if you go through the trouble of writing a foundational proof of safety of your code, you would rather not trust an unverified conservative garbage collector (GC) with your data. For this reason, it is important to be able to write a type-safe GC, but the state of the art in this matter is still completely impractical: it cannot even handle cyclic data-structures. This chapter's main goals are thus:

- Argue that, in order to type-check a GC that can collect cyclic data-structures, the type system has to provide a form of assignment that can change the type of a location (i.e. a *strong update*) even if the set of aliases to this location is unknown.
- Present a type system that provides such a facility. This type system allows the programmer to choose any mix of linear or intuitionistic typing of references and to seamlessly change this choice over time to adapt it to the current needs.
- Implement a type-preserving GC that properly collects cycles and a generational GC that allows the mutator to use destructive assignment.

Traditional type systems are not well-suited to reason about type safety of low-level memory management such as explicit memory allocation, initialization, deallocation, or reuse. Existing solutions to these problems either have a very limited applicability or rely on some form of *linearity* constraint. Such constraints tend to be inconvenient and a lot of work has gone into relaxing them. For example, the alias-types system [WM00] is able to cleanly handle several of the points above, even in the presence of arbitrary aliasing, as long as the aliases can be statically tracked by the type system.

The reason why it is challenging to show type safety of low-level memory management is that for this kind of code, the line between *type safety* and *correctness* is blurred: we end up having to prove some non-trivial properties about the code just to show its type safety. For example, type safety of a generational GC depends on the correct processing of the remembered-set (a data-structure holding the set of pointers from the old generation to the new).

An alternative approach would be to use Hoare logic [Hoa69] to show the correctness of the low-level code and then provide a type-safe interface to it. But it is not clear how those two parts would interact: the low-level code might be spread in pieces over a lot of code and might need to propagate complex invariants to the various pieces through the type-safe interface, as is the case for the code that maintains the remembered-set in the mutator. Furthermore as we start to encode more properties than basic type safety into our type systems, the difficulties we are seeing here will start to appear for more mundane code as well. This tendency can already be seen in the Vault project which uses an approach taken from alias-types to prove other properties of their code than just type safety.

The present work is thus an attempt to provide a middle ground between Hoare logic and traditional type systems. Additionally to the above stated goals, we make the following contributions:

- A language that subsumes traditional region calculi as well as alias-types calculi to simultaneously combine the benefits of traditional *intuitionistic* references and *linear* references.

- We introduce type cast on memory locations and strong update operations that work in the absence of any static aliasing information.

- The language also offers the ability to iterate over the objects contained in a sequential area of memory.
- We show how to use the calculus of inductive constructions (CiC) to track properties of state. This extends the work of Shao et al. [SSTP02] where they used CiC as their type language to track arbitrary properties of values.

Section 5.1 gives a quick preview of the basic idea developed in this chapter. Section 5.2 introduces the problem of cyclic data-structures as well as two type systems on which our work is built. Section 5.3 describes the core part of the new language. Section 5.4 shows some examples of what the language can do. Section 5.5 extends the language with several features needed by Sec. 5.6 where we show how we use it to write a generational GC. We then discuss related work and conclude. Additionally Sec. A.2 shows the soundness proofs of both the core and the extended language.

5.1 Overview

The typed regions system presented in this chapter is basically a hybrid between traditional region systems and alias types systems. In a traditional region system, the type of a pointer completely determines the type of the object to which it points. In alias types systems on the other hand, the type of the pointer does not carry any information about the type of the object to which it points. Instead, the type of the pointer only indicates the location to which it points, and a separate environment is used to look up the type of that location. Typed regions combines those two such that the type of a location is partly determined by the type of the pointers and partly by a separate environment.

The key idea is to introduce the concept of the *intended type* of a memory location, which stays constant throughout the lifetime of that location and thus corresponds to a traditional (intuitionistic) type, supplemented with a non-constant map that translates the intended type of each location to its *actual type*.

The intended type of a location is typically a high-level view of the type of objects that it can hold, that abstracts away time-varying details such as the fact that some fields might be temporarily uninitialized, or that the object is currently replaced by a forwarding pointer.

Let's say we have a pointer of type τ `at` $\rho.n$: that means that it points to an object of intended type τ at location n in region ρ. Each region has a type φ. This type is a function that takes two arguments, the location n of an object and its intended type τ, and returns the actual type of the object $\varphi\ n\ \tau$.

Because the intended types are reflected in the types of pointers, they are kept immutable, since changing the intended type of a location would require updating the type of all the pointers to that location. On the other hand, assignments can change the actual type of a location by modifying the region's type φ.

If φ is a simple identity function that ignores n, then intended types and actual types are the same and we have basically a traditional region system. On the other hand, if φ ignores τ and only uses n to determine the actual type of a location, we have a system reminiscent of alias types.

The difference in power between typed regions and alias types is similar to the difference between destructive update and a simulation of it using functional update: when functionally updating one element of a data-structure, one needs to rebuild all the spines that lead to this element which requires one to keep track of those spines, whereas with destructive updates, the operation can be done without any knowledge of where this object is currently referenced.

5.2 Background

5.2.1 Cyclic structures

In the course of writing the *copy* routine of a garbage collector, we discovered that although current type systems can handle the case where the graph is acyclic, generalizing the code to properly handle cycles proves difficult. After experimenting with various algorithms, it became clear that the problem is more fundamental: current type systems are unable to type-check some generic code that can build arbitrary cyclic data-structures. By generic, I mean that it can apply to objects of any type. In other contexts, it could be called *polytypic*, or *intentionally polymorphic*. The most obvious examples are *gccopy* and *unpickle*.

To see this, let us look at a classic example, a datatype for doubly-linked lists:

$$\textsf{datatype}\ \alpha\ \mathit{dlist} = \mathit{Node}\ \textsf{of}\ \alpha * \alpha\ \mathit{dlist}\ \mathit{ref} * \alpha\ \mathit{dlist}\ \mathit{ref}$$

The SML type system allows us to declare this datatype and write functions to manipulate it, but does not offer us any way to create such an object because there is no base case to start from. But even if we have a base case, the type system can get in the way. Let us take another example:

$$\textsf{datatype}\ \alpha\ \mathit{tree} = \mathit{Node}\ \textsf{of}\ \alpha \mid \mathit{Branch}\ \textsf{of}\ \alpha\ \mathit{tree}\ \mathit{ref} * \alpha\ \mathit{tree}\ \mathit{ref}$$

This time, we can construct such trees since we do have a base case, but if we want to construct an "infinite" tree with no actual *Node* in it, we still first need to build a *Node*. More specifically, in order to create a cyclic data-structure, we always need a base case to start from, even if the data-structure we want to get in the end does not contain any node of this base case any more. But in the above example, we can only build a base case if we have an object of the proper type α. Which means that a generic routine such as an unpickler or a copying GC needs to be able to construct from scratch the base case of any type that could be involved in a cyclic data-structure. In the tree example above, that means creating a Node $\textsf{of}$ α for any α. Clearly this is not possible.

OCaml provides special support to build cyclic data-structures, such as the *dlist* example above, with $\textsf{val rec}\ n = \mathit{Node}(0, n, n)$. This helps for specific code, but is of no use for generic code since it only works for pre-determined cycles, whereas a copying GC simply does not even know when it is creating cycles.

$$
\begin{array}{lrl}
(\textit{kinds}) & \kappa & ::= \Omega \mid \mathsf{R} \\
(\textit{regions}) & \rho & ::= r \mid \nu \\
(\textit{kind env}) & \Delta & ::= \bullet \mid \Delta, t\!:\!\kappa \\
(\textit{type env}) & \Gamma & ::= \bullet \mid \Gamma, x\!:\!\sigma \\
(\textit{heap type}) & \Psi & ::= \bullet \mid \Psi, \nu.n \mapsto \sigma \\
(\textit{region env}) & \Theta & ::= \bullet \mid \Theta, \rho \\
\\
(\textit{types}) & \sigma & ::= t \mid \mathsf{int} \mid \sigma \times \sigma \mid \sigma \ \mathsf{at}\ \rho \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 \\
\\
(\textit{values}) & v & ::= x \mid i \mid (v, v) \mid \nu.n \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
(\textit{operations}) & op & ::= v \mid \pi_i v \mid \mathsf{put}[\rho]\ v \mid \mathsf{get}\ v \\
(\textit{terms}) & e & ::= v[\vec{\sigma}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = op\ \mathsf{in}\ e \mid \mathsf{set}\ v := v; e \\
& & \quad \mid \mathsf{let}\ r = \mathsf{newrgn}\ \ \mathsf{in}\ e \mid \mathsf{freergn}\ \rho; e
\end{array}
$$

Figure 5.1: Syntax of a region-based language.

Type systems that can decouple allocation from initialization are the key to solving this problem, but none of the systems developed so far are sufficiently flexible to handle the case of a generic function such as *unpickle*. More specifically, none of them know how to handle the case where the pointer to the allocated object *escapes* (i.e. is passed around and stored at arbitrary locations) before the object is initialized: when we allocate a new object, we obviously know its one and only alias, but we cannot fully initialize yet it because some of the values might not exist yet, and by the time we are done unpickling the children such that initialization can complete, there can be any number of aliases and we do not statically know them because the function is generic.

In order to type-check a practical copying GC, we need a new type system that is able to update the type (e.g. from uninitialized to initialized) of all the aliases to a particular object even when those aliases are not statically known.

5.2.2 Regions

Region-based type systems [TT94, CWM99] are the most practical systems offering type-safe explicit memory management. They provide a solution to the problem of safe deallocation, with a minimum of added constraints. Even though they do not offer any help when trying to type-check low-level code such as object initialization, their practicality makes them very attractive as a starting point. The idea behind region calculi is to only provide bulk deallocation of a whole region (group of objects) at a time. This way the type system only needs to keep track of regions rather than individual objects: the type of every pointer is simply annotated with the region that it references.

Figure 5.1 an example of such a language. $\mathsf{put}[\rho]\ v$ allocates v in region ρ; $\mathsf{get}\ v$ dereferences v and returns the object it points to; newrgn creates a new region; $\mathsf{freergn}$ deallocates it; $\mathsf{set}\ v_1 := v_2$ places v_2 at the location to which v_1 points; $\nu.n$ is a pointer to the n^{th} object created by put in region ν and has type σ at ν if the object referenced has type σ; $\lambda[\Delta]\{\Theta\}(\Gamma).e$ is a function of type $\forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0$; functions are fully closed and use continuation passing style, so they never return (hence the $\to 0$ in the type); Δ is the list of type parameters; Θ lists the regions that need to be live at the time of the call; and $\vec{\sigma}$ and Γ lists the types of the value parameters. A function call $v[\vec{\sigma}](\vec{v})$ passes types $\vec{\sigma}$ and values $\vec{v}$ to function v. Type variables can have two kinds, depending on whether they range over the types σ of kind Ω of objects or over the types ρ of kind R of regions; Ψ does not appear in the terms but is used in the typing rules (not shown here) where it keeps track of the type of each memory location, such that for all reference value $\nu.n$ its type is $\Psi(\nu.n)$ at ν.

Region calculi do not all look like the above, of course. They do not all use fully closed functions and continuation passing style, for example, but a direct-style presentation would be more complex, as is the correct treatment of closure allocation. Also this language is quite different from the one used in the previous chapter where we had only instead of $\mathsf{freergn}$. The reason is that a calculus based on only made the presentation simpler but it does not lend itself to the kind of extension that are presented in this chapter.

Here is a sample function that creates a cyclic node of the *tree* datatype presented

$$\begin{array}{lrcl}
(\textit{kinds}) & \kappa & ::= & \Omega \mid \mathsf{Heap} \mid \mathsf{Loc} \\
(\textit{locations}) & \rho & ::= & r \mid \nu \\
(\textit{type env}) & \Gamma & ::= & \bullet \mid \Gamma, x\!:\!\sigma \\
(\textit{kind env}) & \Delta & ::= & \bullet \mid \Delta, t\!:\!\kappa \\
(\textit{mem env}) & \Theta & ::= & \bullet \mid \Theta, \rho \mapsto (\sigma, ..., \sigma) \\
\\
(\textit{types}) & \sigma & ::= & t \mid \mathsf{int} \mid \rho \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 \\
\\
(\textit{values}) & v & ::= & x \mid i \mid \nu \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
(\textit{operations}) & op & ::= & v \mid \pi_i v \\
(\textit{terms}) & e & ::= & v[\vec{\sigma}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = op\ \mathsf{in}\ e \mid \mathsf{free}\ \rho; e \\
 & & & \mid \mathsf{let}\ (r, x) = \mathsf{new}\ n\ \mathsf{in}\ e \mid \mathsf{set}\ \pi_i v := v; e
\end{array}$$

Figure 5.2: Syntax of an alias-types language.

previously, assuming the language has been extended with support for datatypes:

$$\begin{array}{l}
\boldsymbol{mktree}[r,t]\{r\}(x\!:\!t, k : \forall[r,t]\{r\}((\mathit{tree}\ t)\ \mathsf{at}\ r) \to 0) \\
= \mathsf{let}\ y = \mathsf{put}[r]\ (\mathit{Node}\ x)\ \mathsf{in} \\
\quad \mathsf{set}\ y := \mathit{Branch}\ y\ y; k[r,t](y)
\end{array}$$

The function expects two type arguments r and t, it expects the region r to be live, and expects an argument x of type t (which is only used temporarily to create the dummy *Node*) and a continuation argument k. The (omitted) kind of r is R and the kind of t is Ω. The put operation allocates memory and temporarily puts a dummy *Node* into it, while the set operation creates the actual cycle. The continuation k also expects two types r and t, it also expects region r to be live and expects a single value argument which is a pointer to a tree in region r. If k's type had $\{\}$ in place of $\{r\}$, it would force us to deallocate the region r before calling it and it would make n into a dangling pointer, which is allowed because liveness of the region is only needed and checked when dereferencing with get.

5.2.3 Alias types

The alias-types system [SWM00, WM00] was developed precisely to handle low-level code such as object initialization, memory reuse, and safe deallocation at the object level. To do that, the type of pointers is changed to carry no information about the type of the referenced object. Instead, the type of a pointer is just the location it is pointing to, so it does not need to change when the location's type or liveness changes. While it provides a lot of power when dealing with low-level code, it relies on an amount of static information which is not always available and definitely not available in our copying GC: if we had this information, we could also statically decide when to deallocate, so we would not need a GC in the first place.

Figure 5.2 shows the syntax of a very simple alias-types language. It can be thought of as a region-based language where the pointers can only point to regions rather than to objects inside them, where regions have been turned into tuples, and where objects inside regions are instead just fields of those tuples. put has disappeared since you cannot add fields to a tuple; get is replaced by π_i; set now only mutates a field of a tuple; pointers ν now just have type ν. The environment Ψ mapping locations to their types has been merged into Θ. When dereferencing a pointer of type ρ, we thus have to check the liveness and the type of the corresponding location by looking up ρ in Θ. $\mathsf{let}\ (r, x) = \mathsf{new}\ n\ \mathsf{in}\ e$ allocates a new object of size n and returns the location as both a value x and a type r. We could also have done this for regions so as to distinguish between the region type and the region value passed to put at runtime, but we conflated the two for simplicity.

Here is a sample code that takes a value of type t and creates an infinite list of this element (a 1-element circular list):

$$
\begin{aligned}
&\boldsymbol{mklist}[\epsilon, t]\{\epsilon\}(x : t, k : \forall[\epsilon, t, r]\{\epsilon, r \mapsto (t, r)\}(r) \to 0) \\
&= \mathsf{let}\ (r, n) = \mathsf{new}\ 2\ \mathsf{in} \\
&\quad \mathsf{set}\ \pi_0 n := x; \mathsf{set}\ \pi_1 n := n; k[\epsilon, t, r](n)
\end{aligned}
$$

The function expects two arguments ϵ and t where ϵ has kind Heap; it expects also that ϵ is live, and it expects to value arguments x of type t and k, the continuation. The type of the continuation shows that it expects three type arguments, where r is a location; it expects the heap ϵ to be live and extended with a pair at location r holding the infinite list; and it expects a single value argument which is the pointer to that list. Since new only knows about the size of the object, it can only do allocation and the type at location r is originally set to $(\mathsf{int}, \mathsf{int})$ and is then incrementally updated by each set operation to (t, int) and then (t, r).

The ability to update a location's type is the key power of alias-types. But for that it relies crucially on the fact that the type system keeps track of pointer values. In particular, the types need to statically but precisely describe the shape of the heap. Witness the fact in the above example that the type of the circular list is not just *list* t but instead explicitly describes a 1-element cycle and thus disallows any other shape. The type language of [WM00] is of course much richer than what we show here, providing a lot more flexibility in the kind of heap shapes you can describe.

5.2.4 Calculus of inductive constructions

The calculus of inductive constructions (CiC) [Pau93] that we use as our type language is an extension of the calculus of constructions (CC) [CH88], which is a higher-order typed logic. CC can encode Church's higher-order predicate logic via the Curry-Howard isomorphism [How80]. Understanding the details of this language is not necessary for this chapter, so I will only give a brief overview, starting with its syntax as a pure type system [Bar91]:

$$
\begin{aligned}
(\mathit{sort})\ s &::= \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\mathit{ptm})\ \varphi &::= s \mid x \mid \lambda x\!:\!\varphi.\,\varphi \mid \varphi\ \varphi \mid \Pi x\!:\!\varphi.\,\varphi \\
&\quad\mid \mathsf{Ind}(x\!:\!\varphi)\{\vec{\varphi}\} \mid \mathsf{Ctor}\,(i, \varphi) \mid \mathsf{Elim}[\varphi](\varphi)\{\vec{\varphi}\}
\end{aligned}
$$

CC is a typed λ-calculus. x is a variable; $\varphi_1\ \varphi_2$ is a function application; $\lambda x\!:\!\varphi_1.\,\varphi_2$ is a function; $\Pi x\!:\!\varphi_1.\,\varphi_2$ is the type of a function taking an argument of type φ_1 and returning a value of type φ_2. This is called a dependent product type and subsumes both the usual function type $\varphi_1 \to \varphi_2$ and the universal quantifier $\forall x\!:\!\varphi_1.\varphi_2$. When the bound variable x does not occur in φ_2, it can be abbreviated $\varphi_1 \to \varphi_2$.

CiC, extends CC with inductive definitions, which can be thought of as ML datatypes.

For example, the following introduces an inductive definition of polymorphic lists:

$$\begin{aligned}&\mathsf{Ind}(\mathit{List}{:}\mathsf{Kind} \to \mathsf{Kind})\\ &\quad \{\mathit{nil} : \Pi x.\mathit{List}\ x \mid \mathit{cons} : \Pi x.x \to \mathit{List}\ x \to \mathit{List}\ x\}\end{aligned}$$

Lists can then be built using the constructor $\mathsf{Ctor}\,(i, \mathit{List})$ where i is either 0 for *nil* or 1 for *cons*. The elimination construct Elim combines case analysis with a fixpoint operation. Objects of an inductive type can thus be iterated over using these constructs. When applied to a list, it behaves like fold. For example, if x is a list, $\mathsf{Elim}[\mathsf{Nat}](x)\{0; \lambda y.\lambda z.z{+}1\}$ returns its length. CiC has been shown to be strongly normalizing [Wer94], hence the corresponding logic is consistent. It is supported by the Coq proof assistant [HPM$^+$00], which I used to experiment with a prototype system of the results presented in this chapter.

In the remainder of this chapter, I will often use more familiar mathematical notation rather than the strict definition of CiC syntax given in this section. For example, inductive definitions will mostly be presented in BNF format. We will, however, retain the Π notation, which can generally be read as a universal quantifier.

5.3 Typed regions

Our new system of typed regions can be thought of as a hybrid between alias-types and the calculus of capabilities [CWM99] supplemented with the calculus of inductive constructions (CiC), similarly to λ_H [SSTP02]. Where alias-types rely on a *linear* map of live locations's types and the calculus of capabilities relies on a linear set of live regions, we rely on a linear map of regions's types.

In a typical region calculus, the type of the object reachable from a pointer (its target) is entirely given by the type of the pointer. In contrast, in the alias-types system, the type of the pointer does not provide any direct information about the type of the target; instead, the target's type is kept in a linearly managed *type map* indexed by the pointer's type, which is the singleton type holding the object's location. Our new type system mixes the

$$
\begin{array}{llcl}
(\mathit{sort}) & s & ::= & \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\mathit{ptm}) & \varphi, \tau, \kappa, P & ::= & s \mid x \mid \lambda x\!:\!\varphi.\,\varphi \mid \varphi\ \varphi \mid \Pi x\!:\!\varphi.\,\varphi \\
 & & & \mid \mathsf{Ind}(x\!:\!\varphi)\{\vec{\varphi}\} \mid \mathsf{Ctor}\,(i, \varphi) \mid \mathsf{Elim}[\varphi](\varphi)\{\vec{\varphi}\} \\
(\mathit{memory}) & M & ::= & \bullet \mid M, \nu.n \mapsto v \\
(\mathit{heap\ type}) & \Psi & ::= & \bullet \mid \Psi, \nu.n \mapsto \tau \\
(\mathit{type\ env}) & \Gamma & ::= & \bullet \mid \Gamma, x\!:\!\sigma \\
(\mathit{kind\ env}) & \Delta & ::= & \bullet \mid \Delta, t\!:\!\kappa \\
(\mathit{region\ env}) & \Theta & ::= & \bullet \mid \Theta, \rho \mapsto (\varphi, n) \\
(\mathit{regions}) & \rho\!:\!\mathsf{R}\ \kappa & ::= & r \mid \nu \\
(\mathit{types}) & \sigma\!:\!\Omega & ::= & t \mid \mathsf{int} \mid \sigma \times \sigma \mid \tau\ \mathsf{at}\ \rho.n \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 \\
(\mathit{values}) & v & ::= & x \mid i \mid (v, v) \mid \nu.n \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
(\mathit{operations}) & op & ::= & v \mid \pi_i v \mid \mathsf{get}\ v \\
(\mathit{terms}) & e & ::= & v[\vec{\varphi}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = op\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{put}[\rho, \tau]\ v\ \mathsf{in}\ e \\
 & & & \mid \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e \mid \mathsf{freergn}\ \rho; e \mid \mathsf{set}\ v \overset{\varphi}{:=} v; e \\
 & & & \mid \mathsf{cast}[P]\ \nu \mapsto \varphi; e
\end{array}
$$

Figure 5.3: Syntax of the core language.

two, such that the pointer's type holds both the location and some information (called the *intended type*) about the object to which it points, while the remaining information is kept in a map of regions' types. The type of a region is a function that maps an object's location and its intended type to its actual type.

5.3.1 The core language

The syntax of a trimmed down version of our language is shown in Fig. 5.3. The language uses continuation passing style and fully closed functions. $M, \Psi, \Gamma, \Delta, \Theta$ are environments used in the typing rules and operational semantics. Values v can be integers, pairs $((v, v)\!:\!\sigma \times \sigma)$, references $(\nu.n\!:\!\tau\ \mathsf{at}\ \nu.n)$, and functions. The terms do the following:

$\mathsf{halt}\ v$	halt the machine, returning v as the result.
$\pi_i v$	select i^{th} from tuple v.
$v[\vec{\varphi}](\vec{v})$	make a tail-call to function v.
$\mathsf{newrgn}\ \varphi$	allocate a new region of type φ.
$\mathsf{freergn}\ \rho; e$	free the region ρ.
$\mathsf{put}[\rho, \tau]\ v$	allocate an object v of intended type τ in region ρ.
$\mathsf{get}\ v$	fetch the object pointed to by v.
$\mathsf{set}\ v_1 \overset{\varphi}{:=} v_2; e$	update location v_1 with value v_2. φ is the new type of the location.
$\mathsf{cast}[P]\ \rho \mapsto \varphi; e$	Set the type of region ρ to φ. P is a proof that φ is a valid replacement.

The language is very much similar to the simple region calculus presented before, except for the following differences:

- The type language is CiC. The kinds such as Ω (the kind of types σ) and R (the kind of regions ρ) along with the basic type constructors of σ are now defined directly as inductive definitions in CiC.

- The region environment Θ now contains not only a list of live regions, but a map from live regions to their type φ and size n. The type φ is a CiC function that maps the index of a location and its *intended type* to the actual type of that location.

- Pointer types have the form τ at $\rho.n$ rather than just σ at ρ, where n is the offset inside the region. Such a pointer points to an object of *intended type* τ but whose actual type can only be discovered by an indirection through the region's type: the target's type is $\varphi\ n\ \tau$ where $\Theta(\rho) = (\varphi, n)$.

- Just as before, a value $\nu.n$ has type $(\Psi(\nu.n))$ at $\nu.n$ but $\Psi(\nu.n)$ is an *intended type* and can be of any kind rather than only Ω.

- The region kind R now takes a parameter κ specifying the kind of intended types in this region. If $\rho\!:\!\mathsf{R}\ \kappa$, then the region's type φ will be of kind $\mathsf{Nat} \to \kappa \to \Omega$ and the term τ in τ at $\rho.n$ will have to have kind κ.

$$
\begin{array}{ll}
(M;\Theta;\Psi;\mathsf{let}\ x = v\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;\Psi;e[{}^{v}/_{x}]) \\
(M;\Theta;\Psi;\mathsf{let}\ x = \pi_i(v_1,v_2)\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;\Psi;e[{}^{v_i}/_{x}]) \\
\begin{array}{l}(M;\Theta;\Psi;\nu.n[\vec{\varphi}](\vec{v})) \\ \quad \text{where } M(\nu.n) = \lambda[\overrightarrow{t:\kappa}]\{\Theta'\}(\overrightarrow{x:\sigma}).e\end{array} & \Longrightarrow (M;\Theta;\Psi;e[{}^{\vec{\varphi},\vec{v}}/_{\vec{t},\vec{x}}]) \\
(M;\Theta;\Psi;\mathsf{let}\ x = \mathsf{get}\ \nu.n\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;\Psi;e[{}^{M(\nu.n)}/_{x}]) \\
\begin{array}{l}(M;\Theta,\nu \mapsto (\varphi,n);\Psi \\ \quad ;\mathsf{let}\ x = \mathsf{put}[\nu,\tau]\ v\ \mathsf{in}\ e)\end{array} & \Longrightarrow \begin{array}{l}(M,\nu.n \mapsto v;\Theta,\nu \mapsto (\varphi,n+1) \\ \quad ;\Psi,\nu.n \mapsto \tau;e[{}^{\nu.n}/_{x}])\end{array} \\
(M;\Theta,\nu \mapsto (\varphi,n);\Psi;\mathsf{freergn}\ \nu;e) & \Longrightarrow (M\backslash\nu;\Theta;\Psi\backslash\nu;e) \\
\begin{array}{l}(M;\Theta;\Psi;\mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e) \\ \quad \text{where } \bullet \vdash^{CiC} \varphi : \mathsf{Nat} \to \kappa \to \Omega\end{array} & \Longrightarrow \begin{array}{l}(M;\Theta,\nu \mapsto (\varphi,0);\Psi;e[{}^{\nu}/_{r}]) \\ \quad \text{where } \nu \text{ is fresh and } \bullet \vdash^{CiC} \nu : \mathsf{R}\ \kappa\end{array} \\
(M;\Theta,\nu \mapsto (\varphi,n);\Psi;\mathsf{cast}[P]\ \nu \mapsto \varphi';e) & \Longrightarrow (M;\Theta,\nu \mapsto (\varphi',n);\Psi;e) \\
(M;\Theta,\nu \mapsto (\varphi',n');\Psi;\mathsf{set}\ \nu.n \overset{\varphi}{:=} v;e) & \Longrightarrow (M,\nu.n \mapsto v;\Theta,\nu \mapsto (upd\ \varphi'\ n\ \varphi,n');\Psi;e)
\end{array}
$$

Figure 5.4: Operational semantics of the core language.

- put takes an additional parameter τ and returns a pointer of type τ at $\rho.n$ after checking that $v{:}\varphi\ n\ \tau$.
- set is now a strong update: it changes the type of the location to φ which has kind $\kappa \to \Omega$. This type needs to be provided because there are many valid choices and they are not all equivalent.
- newrgn now takes a parameter φ which is the initial type of the region.

We have decided to manipulate whole objects rather than words and not to split allocation from initialization. It is easy to change the language to provide alloc, load, and store instead of put, get, and set, but the typing rules become more verbose and so does the code in the examples.

5.3.2 Semantics

Figure 5.4 shows the typed operational semantics. The machine state is defined as the 4-tuple $(M;\Theta;\Psi;e)$. The memory type Ψ can basically be ignored in those reduction steps, since it will only really play a role when we extend the language in Sec. 5.5. Also, the region

$\vdash \Psi$	memory type Ψ is well-formed
$\vdash \Delta$	type environment Δ is well-formed
$\Delta \vdash \Theta$	region type environment Θ is well-formed in context Δ
$\Delta \vdash \Gamma$	value environment Γ is well-formed in context Δ
$\Psi; \Theta \vdash M$	memory M is well formed in Ψ and Θ
$\vdash (M; \Theta; \Psi; e)$	machine state is well-formed

$$\frac{\bullet \vdash^{CiC} \varphi_{ij} : \kappa \qquad \bullet \vdash^{CiC} \nu_i : \mathsf{R}\ \kappa}{\vdash \nu_0.0 \mapsto \varphi_{00}, \ldots, \nu_m.n \mapsto \varphi_{mn}} \qquad \frac{}{\vdash \bullet} \qquad \frac{\vdash \Delta \qquad \Delta \vdash^{CiC} \kappa : s}{\vdash \Delta, t{:}\kappa}$$

$$\frac{}{\Delta \vdash \bullet} \qquad \frac{\begin{array}{c}\Delta \vdash \Theta \qquad \rho \notin \mathit{Dom}(\Theta) \qquad \Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa \\ \Delta \vdash^{CiC} \varphi : \mathsf{Nat} \to \kappa \to \Omega \qquad \Delta \vdash^{CiC} n : \mathsf{Nat}\end{array}}{\Delta \vdash \Theta, \rho \mapsto (\varphi, n)} \qquad \frac{\Delta \vdash^{CiC} \sigma_i : \Omega}{\Delta \vdash x_0{:}\sigma_0, \ldots, x_n{:}\sigma_n}$$

$$\frac{\forall i{:}0..n\ .\ \forall j{:}\mathsf{Nat}\ .\ \text{if } j \geq n_i \text{ then } \nu_i.j \notin \mathit{Dom}(\Psi) \text{ else } \Psi; \bullet; \Theta; \bullet \vdash M(\nu_i.j) : \varphi_i\ j\ (\Psi(\nu_i.j))}{\Psi; \Theta = \nu_0 \mapsto (\varphi_0, n_0), \ldots, \nu_n \mapsto (\varphi_n, n_n) \vdash M}$$

$$\frac{\Psi; \Theta \vdash M \qquad \bullet \vdash \Theta \qquad \vdash \Psi \qquad \Psi; \bullet; \Theta; \bullet \vdash e}{\vdash (M; \Theta; \Psi; e)}$$

Figure 5.5: Environment formation rules.

environment Θ is only marginally used in put where we need to find the next free location in the region. After type erasure, Ψ would disappear and Θ would only keep track of the size of each region. The rules use the auxiliary type function *upd* which takes a function φ taking an argument of type Nat and returns a function equal to it, except at point n where it now returns φ' instead:

$$\mathit{upd}\ \varphi\ n\ \varphi' = \lambda i.\mathsf{if}\ (i = n)\ (\varphi')\ (\varphi\ i)$$

The reduction rules are pretty unexceptional. Things to note are that cast is indeed a no-op which does not affect anything other than types. Also the type region environment after a function call is Θ rather than Θ': the two should be equivalent and e has been type-checked using Θ', but the use of Θ makes it more obvious that the operation is just a jump. Finally the additional type constraints on the reduction rule for newrgn are only present to simplify the presentation of the soundness proof and would disappear after type erasure.

The formation rules for environments are given in Fig. 5.5 together with the definition of a well-formed machine state $\vdash (M; \Theta; \Psi; e)$. A few notes about the notation used in the rules: environments on the left-hand side of a $\vdash$ are treated as unordered even for Δ and Θ where order matters. I.e. $\Delta, t\!:\!\kappa \vdash \Gamma$ is actually a shorthand for $\Delta', t\!:\!\kappa, \Delta'' \vdash \Gamma$ where $\Delta = \Delta', \Delta''$. Also, when an i appears in the premises of a rule, it is universally quantified. Finally, a meta-variable $\vec{x}$ is completely separate from a meta-variable x and is equivalent to a list $x_0..x_n$.

The judgment $\Delta \vdash^{CiC} \varphi : \kappa$ used in those rules, taken directly from CiC and not shown here, states that φ has kind κ in environment Δ. The judgment $\vdash \Delta$ is also taken directly from CiC. The judgment $\vdash \Psi$ checks that each intended types has a kind consistent with its region. The judgment $\Delta \vdash \Gamma$ checks that each variable's type has kind Ω. The judgment $\Delta \vdash \Theta$ checks that the type of each region is consistent with its kind and that each region has only one binding in Θ (a linearity constraint). This is important because in order to be able to free region ρ or to change its type, we need to be sure that the same physical region is not referred to somewhere else under some other name. The judgment $\Psi; \Theta \vdash M$ is the one that expresses the invariant that needs to hold so that intended types, actual types and region types are all consistent with one another. It checks that the actual type of each object in M indeed matches the result of applying to its intended type (stored in Ψ) the corresponding region's type (stored in Θ). In a simple region system, the well-formedness of M is sometimes written as $\vdash M : \Psi$ so in our case the judgment $\Psi; \Theta \vdash M$ could be thought of as $\vdash M : \Theta(\Psi)$ where Θ is taken as a function that interprets the intended memory type Ψ and returns an actual memory type. An important detail about the rule is that it verifies that Ψ has no binding for not-yet-allocated locations. This is needed because the intended type is immutable, so Ψ can only be extended but none of its existing bindings can be modified.

Figure 5.6 shows the formation rules for types and terms. The equivalence rule $\Theta \triangleleft \Theta'$ is used to formalize the fact that Θ is not ordered. In the rule for pointer values $\nu.n$, dangling pointers to dead regions can have any type (because the rule for get prevents dereferencing

$$\begin{array}{|l|l|}\hline \Delta \vdash \vec{\sigma} : \overrightarrow{t{:}\kappa} & \text{actual type arguments } \vec{\sigma} \text{ match formal arguments } \overrightarrow{t{:}\kappa} \\ \Theta \sim \Theta' & \Theta \text{ is equivalent to } \Theta' \\ \hline \end{array}$$

$$\frac{\Delta \vdash^{CiC} \sigma : \kappa \qquad \Delta \vdash \vec{\sigma} : \overrightarrow{t{:}\kappa}[{}^{\sigma}/_{t}]}{\Delta \vdash \sigma, \vec{\sigma} : t{:}\kappa, \overrightarrow{t{:}\kappa}} \qquad \frac{}{\Theta \sim \Theta} \qquad \frac{\Theta \sim \Theta'}{\Theta, \rho \mapsto (\varphi, n) \sim \Theta', \rho \mapsto (\varphi, n)}$$

$$\begin{array}{|l|l|}\hline \Psi; \Delta; \Theta; \Gamma \vdash v : \sigma & \text{value } v \text{ has type } \sigma \\ \Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \sigma & \text{value } v \text{ is a pointer to an object of type } \sigma \\ \Psi; \Delta; \Theta; \Gamma \vdash \mathit{op} : \sigma & \text{operation } \mathit{op} \text{ returns a value of type } \sigma \\ \hline \end{array}$$

$$\frac{}{\Psi; \Delta; \Theta; \Gamma \vdash n : \mathsf{int}} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v_i : \sigma_i}{\Psi; \Delta; \Theta; \Gamma \vdash (v_1, v_2) : \sigma_1 \times \sigma_2} \qquad \frac{}{\Psi; \Delta; \Theta; \Gamma \vdash x : \Gamma(x)}$$

$$\frac{\nu \in \mathit{Dom}(\Theta) \Rightarrow \tau = \Psi(\nu.n)}{\Psi; \Delta; \Theta; \Gamma \vdash \nu.n : \tau \ \mathsf{at}\ \nu.n} \qquad \frac{\vdash \Delta \qquad \Delta \vdash \Theta \qquad \Delta \vdash \overrightarrow{x{:}\sigma} \qquad \Psi; \Delta; \Theta; \overrightarrow{x{:}\sigma} \vdash e}{\Psi; \Delta'; \Theta'; \Gamma \vdash \lambda[\Delta]\{\Theta\}(\overrightarrow{x{:}\sigma}).e : \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0}$$

$$\frac{\begin{array}{c}\Theta(\rho) = (\varphi, m) \\ \Psi; \Delta; \Theta; \Gamma \vdash v : \tau \ \mathsf{at}\ \rho.n\end{array}}{\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \varphi\ n\ \tau} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \sigma}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{get}\ v : \sigma} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma_1 \times \sigma_2}{\Psi; \Delta; \Theta; \Gamma \vdash \pi_i v : \sigma_i}$$

$$\begin{array}{|l|l|}\hline \Psi; \Delta; \Theta; \Gamma \vdash e & \text{expression } e \text{ is well-formed} \\ \hline \end{array}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash \mathit{op} : \sigma \qquad \Psi; \Delta; \Theta; \Gamma, x{:}\sigma \vdash e}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{let}\ x = \mathit{op}\ \mathsf{in}\ e} \qquad \frac{\Psi; \Delta; \bullet; \Gamma \vdash v : \mathsf{int}}{\Psi; \Delta; \bullet; \Gamma \vdash \mathsf{halt}\ v}$$

$$\frac{\begin{array}{c}\Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa \qquad \Delta \vdash^{CiC} \tau : \kappa \qquad \Psi; \Delta; \Theta; \Gamma \vdash v : \varphi\ n\ \tau \\ \Psi; \Delta; \Theta, \rho \mapsto (\varphi, n{+}1); \Gamma, x{:}\tau \ \mathsf{at}\ \rho.n \vdash e\end{array}}{\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash \mathsf{let}\ x = \mathsf{put}[\rho, \tau]\ v\ \mathsf{in}\ e}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \forall[\overrightarrow{t{:}\kappa}]\{\Theta'\}(\vec{\sigma}) \to 0 \\ \Delta \vdash \vec{\varphi} : \overrightarrow{t{:}\kappa} \quad \Theta \sim \Theta'[{}^{\vec{\varphi}}/_{\vec{t}}] \quad \Psi; \Delta; \Theta; \Gamma \vdash v_i : \sigma_i[{}^{\vec{\varphi}}/_{\vec{t}}]\end{array}}{\Psi; \Delta; \Theta; \Gamma \vdash v[\vec{\varphi}](\vec{v})} \qquad \frac{\begin{array}{c}\Delta \vdash^{CiC} \varphi : \mathsf{Nat} \to \kappa \to \Omega \\ \Psi; \Delta, r{:}\mathsf{R}\ \kappa; \Theta, r \mapsto (\varphi, 0); \Gamma \vdash e\end{array}}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e}$$

$$\frac{\begin{array}{c}\Delta \vdash^{CiC} P : \Pi i{:}0..n{-}1.\, \Pi t{:}\kappa.\, (\varphi\ i\ t) = (\varphi'\ i\ t) \\ \Psi; \Delta; \Theta, \rho \mapsto (\varphi', n); \Gamma \vdash e \qquad \Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa\end{array}}{\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash \mathsf{cast}[P]\ \rho \mapsto \varphi'; e} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash e}{\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash \mathsf{freergn}\ \rho; e}$$

$$\frac{\Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa \qquad \begin{array}{c}\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash v : \tau \ \mathsf{at}\ \rho.m \\ \Delta \vdash^{CiC} \varphi' : \kappa \to \Omega \qquad \Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash v' : \varphi'\ \tau \\ \Psi; \Delta; \Theta, \rho \mapsto (\mathit{upd}\ \varphi\ m\ \varphi', n); \Gamma \vdash e\end{array}}{\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash \mathsf{set}\ v \overset{\varphi'}{:=} v'; e}$$

Figure 5.6: Static semantics of the core language.

them), but pointers past the allocation line of a region are disallowed by checking that they have a binding in Ψ. This way, pointers are live iff their region is live. In the rule for functions, the function is forced to be fully closed by typing its body in an environment that does not include Γ or Δ'. The rule for the trivial operation where op is v is not shown but just delegates to the rule for values. The auxiliary rule for $v \mapsto \sigma$ is used by get and by function calls to make sure that the pointer is not dangling and also to find out the actual type at that location. The rule for put checks that the region is live, with appropriate type, and updates the size. The rule for function calls checks that the arguments have the proper kind and type and also that the current region environment is equivalent to the one expected by the function. The rule for cast checks that the region is live and that P indeed proves that the new type σ' is equivalent to the old type for all the live locations and for all possible intended types. It does not pay attention to which intended types are actually used at those locations because those intended types are, in general, not known yet when type-checking. The rule for freergn checks that the region is live before and that the rest of the code does not use the region any more. The rule for set does not use the auxiliary rule $v \mapsto \sigma$ because it does not care about the actual type before assignment, since it will overwrite the location. Instead it just checks that the pointer is live and that the new value matches the new type.

5.3.3 Properties of the core language

We state here a few important properties of the language. The proofs can be found in Sec. A.2. Since our type language is CiC, we know it is strongly normalizing and confluent.

Lemma 5.3.1 (Type Preservation)

If $\vdash (M;\Theta;\Psi;e)$ *and* $(M;\Theta;\Psi;e) \Longrightarrow (M';\Theta';\Psi';e')$ *then* $\vdash (M';\Theta';\Psi';e')$.

Lemma 5.3.2 (Progress)

If $\vdash (M;\Theta;\Psi;e)$, *either* $e = \mathsf{halt}\ v$ *or* $(M;\Theta;\Psi;e) \Longrightarrow (M';\Theta';\Psi';e')$.

Lemma 5.3.3 (Complete Collection)

If $\vdash (M;\Theta;\Psi;e)$ *and* $\forall\nu.n\in Dom(M)\ .\ \nu\in Dom(\Theta)$ *and* $(M;\Theta;\Psi;e) \Longrightarrow (M';\Theta';\Psi';e')$ *then* $\forall\nu.n\in Dom(M')\ .\ \nu\in Dom(\Theta')$*. In particular, if* $e' = \mathsf{halt}\ v$ *then* $M' = \bullet$*.*

Proof The proof follows trivially from inspection of the reduction rules. The particular case of $\mathsf{halt}\ v$ uses additionally the preservation lemma to verify that $\vdash (M';\Theta';\Psi';e')$. □

5.4 Examples

To get an idea of how the language is used, here are some examples of how you can simulate the behavior of other systems in this language.

5.4.1 Simple regions

Given the differences between the simple region system presented before and the typed region system, here is what we have to do to translate a program written in the simple region system:

- Add region types and region sizes to Θ. Assuming that intended types are always the same as actual types, the regions's types will always be $\lambda i.\lambda t.t$. On the other hand their size will keep changing, so it needs to be passed around as extra type argument of kind Nat.

- Add the kind of intended types to region kinds R. Since the intended types are the same as actual types, the region kind is $\mathsf{R}\ \Omega$.

- Annotate put with the type of the allocated object.

- Tell set that the region's type is left unchanged. At first it seems like using $\lambda t.t$ for the type annotation φ is enough. But then the type of the region after set is:

$$\lambda i.\mathsf{if}\ (i = n)\ (\lambda t.t)\ ((\lambda i.\lambda t.t)\ i)$$
$$\Downarrow \text{ which reduces to } \Downarrow$$
$$\lambda i.\mathsf{if}\ (i = n)\ (\lambda t.t)\ (\lambda t.t)$$

where n is the location referenced by the pointer. But this is not equal to $\lambda i.\lambda t.t$, so we need to add a cast to reshape the region's type to what we need:

$$\mathsf{cast}[P]\ \rho \mapsto \lambda i.\lambda t.t; e$$

And P needs to show equivalence between the two types:

$$P : \Pi i\!:\!0..n\!-\!1.\, \Pi t\!:\!\kappa.\, ((\lambda i.\mathsf{if}\ (i = n)\ (\lambda t.t)\ (\lambda t.t))\ i\ t) = ((\lambda i.\lambda t.t)\ i\ t)$$
$$\Downarrow$$
$$P : \Pi i\!:\!0..n\!-\!1.\, \Pi t\!:\!\kappa.\, ((\mathsf{if}\ (i = n)\ (\lambda t.t)\ (\lambda t.t))\ t) = t$$
$$\Downarrow$$
$$P = \lambda i.\lambda t. < \lambda b.((\mathsf{if}\ b\ (\lambda t.t)\ (\lambda t.t))\ t) = t >$$
$$\mathsf{if}\ (i = n)\ (\mathit{refl\_equal}\ t)\ (\mathit{refl\_equal}\ t)$$

Where "$\mathit{refl\_equal}\ \varphi$" is the proof of $\varphi = \varphi$ and where "$<f>\mathsf{if}\ b\ \varphi_1\ \varphi_2$" is a proof of $f\ b$ if φ_1 is a proof of $f\ \mathit{true}$ and φ_2 is a proof of $f\ \mathit{false}$.

- Pointer types $\sigma\ \mathsf{at}\ \rho$ need to be turned into $\sigma\ \mathsf{at}\ \rho.n$, but since we do not know n we have to hide it with an existential package: $\exists n.\sigma\ \mathsf{at}\ \rho.n$. Existential packages will be added in the full language shown further.

For convenience, let's define the two type functions $\mathit{id} = \lambda t.t$ and $\mathit{idr} = \lambda i.\mathit{id}$. The example from the introduction becomes:

$$\boldsymbol{mktree}[r, n, t]\{r \mapsto (\mathit{idr}, n)\}(x\!:\!t, k : \forall[r, n, t]\{r \mapsto (\mathit{idr}, n)\}((\mathit{tree}\ t)\ \mathsf{at}\ r.(n\!-\!1)) \to 0)$$
$$= \mathsf{let}\ y = \mathsf{put}[r, \mathit{tree}\ t]\ (\mathit{Node}\ x)\ \mathsf{in}$$
$$\mathsf{set}\ y \stackrel{\mathit{id}}{:=} \mathit{Branch}\ y\ y; \mathsf{cast}[P]\ r \mapsto \mathit{idr}; k[r, n\!+\!1, t](y)$$

Here I avoided using existential packages by making the actual address of the new tree passed to the continuation k explicit. To make writing such code easier, one would define a new set that does a traditional weak update by coupling the set and the cast above. The burden of keeping track of the allocation limit n is not as bad as it seems since each function is actually a basic block with a fixed number of allocations in it.

5.4.2 Stacks

We can use a typed region to represent a contiguous stack of objects. More specifically, a stack is a region S of kind $\mathsf{R}\ \mathsf{Unit}$. The locations in a stack have no intended type, so we use the dummy Unit kind which has a single element denoted $()$. A function using such a stack will have the following shape:

$$
\begin{array}{l}
\lambda[S\!:\!\mathsf{R}\ \mathsf{Unit}, st\!:\!ST, sp, ss\!:\!\mathsf{Nat}, \dots]\\
\ \{S \mapsto (st, ss), \dots\}\\
\ \left(\begin{array}{l}
sp\!:\!()\ \mathsf{at}\ S.sp\\
k\!:\!\forall[st'\!:\!ST, \dots]\\
\qquad \{S \mapsto (\lambda n.\mathsf{if}\ (n < sp)(st\ n)(st'\ n), ss), \dots\}\\
\qquad (\dots) \to 0\\
\dots
\end{array}\right).\\
\ \dots
\end{array}
$$

The kind of the stack type is defined as $ST = \mathsf{Nat} \to \mathsf{Unit} \to \Omega$ where the second argument, of kind Unit, is unused. The type function st, maps the locations in the stack to their current type. The top of the stack is kept in sp, both as a value and a type. The size of the stack is kept in the type variable ss. The type of the continuation k specifies that the stack S when we return to k has the same type as before for all elements below sp, while the rest can be changed at will and described by the type function st'.

When pushing a new element on the stack, we somehow need to find the address of the next consecutive element in region S. Let us extend the language with such a facility, called $\mathsf{ifnext}\ v\ (x, t\ .\ e_1)\ (e_2)$ that simply takes a pointer v, checks whether it is the last element in the region, if so continues with e_2, otherwise continues with e_1 with x bound to a pointer to the next consecutive element in the region and t bound to its intended type. Now pushing an object on the stack is done by:

$$\mathsf{set}\ sp \stackrel{\lambda\_.\sigma}{:=} v; \mathsf{ifnext}\ sp\ (sp, \_\ .\ e)\ (\mathsf{halt}\ 0)$$

where e is the rest of the computation, _ indicates that we do not care about this argument (it will always be () anyway), σ is the type of v, and where we could do something more clever in case of stack overflow.

Popping elements is simply done implicitly by reverting to a previously saved value of *sp*.

Before returning to the continuation k we need to find an appropriate *st'*, and show that the current type of S is indeed the one expected by the continuation. This proof will be passed to a cast operation just before jumping to k. The first part is trivial since the current stack type is exactly what we need to pass as *st'*. The second part requires proving that $\lambda n.\mathsf{if}\ (n < sp)(st\ n)(st'\ n)$ is equivalent to *st'*. The type of S at that point will have the following shape:

$$\begin{aligned} st' = \lambda n.\text{if } (n = sp)\ &(\lambda\_.\sigma_1) \\ &(\text{if } (n = sp{+}1)\ (\lambda\_.\sigma_2) \\ &\quad(\text{if } \ldots (st\ n))) \end{aligned}$$

The proof mainly entails showing that *st* and *st'* (i.e. the old and the new stack types) are equal w.r.t. locations below *sp*, which is easy to show since pushing only modifies locations above *sp*.

5.4.3 *Adoption* and *focus*

The Vault language [DF01] uses a mix of alias types and traditional types to provide a powerful user-level language used to write device drivers, where the type system is used among other things to check the correct ordering of operations such as open, write, close. They do that by allowing some types to be *tracked*, meaning that they behave linearly like alias types. Operations on objects of tracked types can update their type, so the type system can keep track of their state.

In a subsequent paper [FD02], the authors extended the language with *adoption* which allows to "untrack" a tracked object so it behaves intuitionistically and *focus* which does the reverse. This allows them to use intuitionistic references while at the same time being able to temporarily strengthen them to a linear reference, using the focus construct:

$$\mathsf{let}\ x = \mathsf{focus}\ e_1\ \mathsf{in}\ e_2$$

The expression e_1 has an intuitionistic type, but while inside e_2, the variable x to which it is bound has a linear type and can thus be modified using strong update. To guarantee soundness, they impose the restriction that even though strong update can modify the type of the object referred to by x, its type should be the same at the end of e_2 as it was at its beginning, so the type modification is only temporary. Also any other object in the same region as e_1 is temporarily unavailable.

We can encode something similar. Let's assume we have a region ρ like the ones used in the simple regions example above. Its type is *idr*, i.e. it does not depend on the location of a given object just as is the case in intuitionistic systems. So references will hide their location using an existential wrapper: $\exists n.\sigma\ \mathsf{at}\ \rho.n$. A code equivalent to the focus construct will then look like:

$$\mathsf{let}\ \langle n, x\rangle = \mathsf{open}\ e_1\ \mathsf{in}\ e_2; \mathsf{cast}[P]\ \rho \mapsto \mathit{idr}\ ; ...$$

Where e_1 has type $\exists n.\sigma\ \mathsf{at}\ \rho.n$ and x has thus type $\sigma\ \mathsf{at}\ \rho.n$. Within e_2, strong update can be used at will on x, but before reaching cast the actual type at location n needs to be reset to σ. Remember that focus had the additional constraint that other references to ρ could temporarily not be used. In our case, we can still use them, although we will not be able to do anything useful with them unless we know their aliasing relationship with x.

$$
\begin{array}{llcl}
(\mathit{sort}) & s & ::= & \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\mathit{ptm}) & \varphi, \tau, \kappa, P & ::= & s \mid x \mid \lambda x\!:\!\varphi.\,\varphi \mid \varphi\ \varphi \mid \Pi x\!:\!\varphi.\,\varphi \\
 & & & \mid \mathsf{Ind}(x\!:\!\varphi)\{\vec{\varphi}\} \mid \mathsf{Ctor}\,(i, \varphi) \mid \mathsf{Elim}[\varphi](\varphi)\{\vec{\varphi}\} \\
(\mathit{memory}) & M & ::= & \bullet \mid M, \nu.n \mapsto v \\
(\mathit{heap\ type}) & \Psi & ::= & \bullet \mid \Psi, \nu.n \mapsto \tau \\
(\mathit{type\ env}) & \Gamma & ::= & \bullet \mid \Gamma, x\!:\!\sigma \\
(\mathit{kind\ env}) & \Delta & ::= & \bullet \mid \Delta, t\!:\!\kappa \\
(\mathit{region\ env}) & \Theta & ::= & \bullet \mid \Theta, \rho\mapsto\theta \\
(\mathit{region\ spec}) & \theta & ::= & (\varphi, n) \mid \vec{\rho} \\
(\mathit{regions}) & \rho\!:\!\mathsf{R}\ \kappa & ::= & r \mid \nu \\
(\mathit{types}) & \sigma\!:\!\Omega & ::= & \mathsf{int} \mid \sigma \times \sigma \mid \tau\ \mathsf{at}\ \rho.n \mid \exists t\!:\!\kappa.\sigma \mid \exists r\!\in\!\vec{\rho}.\sigma \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 \\
(\mathit{values}) & v & ::= & x \mid i \mid (v, v) \mid \nu.n \mid \langle \varphi, v\rangle \mid \langle \rho, v\rangle \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
(\mathit{operations}) & \mathit{op} & ::= & v \mid \pi_i v \mid \mathsf{get}\ v \mid \mathsf{cast}[P]\ v \\
(\mathit{terms}) & e & ::= & v[\vec{\varphi}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = \mathit{op}\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{put}[\rho, \tau]\ v\ \mathsf{in}\ e \\
 & & & \mid \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e \mid \mathsf{freergn}\ \rho; e \mid \mathsf{set}\ v \overset{\varphi}{:=} v; e \\
 & & & \mid \mathsf{cast}[P]\ \rho \mapsto \varphi; e \mid \mathsf{let}\ \langle t, x\rangle = \mathsf{open}\ v\ \mathsf{in}\ e \\
 & & & \mid \mathsf{let}\ \langle r, x\rangle = \mathsf{openrgn}\ v\ \mathsf{in}\ e \mid \mathsf{iffirst}\ \rho\ (x, t\ .\ e)\ (e) \\
 & & & \mid \mathsf{ifnext}\ v\ (x, t\ .\ e)\ (e)
\end{array}
$$

Figure 5.7: Syntax of the full language

5.5 The full language

As mentioned, this system solves the three problems we have encountered when trying to code a generational GC: The *before* b test can simply compare b to the offset n of the pointer since it is now carried in its type; Since all objects in a region have a type of the form $\varphi\ n\ t$, setting the region's type to something like C makes sure that first indeed returns a pointer to an object of type $C(t)$; Finally, since Θ is managed linearly and since all memory accesses have to look up the target's type in Θ, we can globally update the type of objects when we move the boundary by updating the corresponding type in Θ.

So here I extend the core calculus by adding various features necessary to write code such as a generational GC. The syntax is shown in Fig. 5.7.

In order to be able to scan regions sequentially, as seen before for stack and as done in GCs as well, we have the following two new primitives:

$\mathsf{iffirst}\ \rho\ (x, t\ .\ e)\ (e)$: test whether ρ is empty and selects the corresponding branch. If

$$
\begin{array}{lll}
(M;\Theta;\Psi;\mathsf{let}\ x=v\ \mathsf{in}\ e) & \Longrightarrow & (M;\Theta;\Psi;e[^{v}/_{x}])\\
(M;\Theta;\Psi;\mathsf{let}\ x=\pi_i(v_1,v_2)\ \mathsf{in}\ e) & \Longrightarrow & (M;\Theta;\Psi;e[^{v_i}/_{x}])\\
\begin{array}{l}(M;\Theta;\Psi;\nu.n[\overrightarrow{\varphi}](\overrightarrow{v}))\\ \quad\text{where } M(\nu.n)=\lambda[\overrightarrow{t\!:\!\kappa}]\{\Theta'\}(\overrightarrow{x\!:\!\sigma}).e\end{array} & \Longrightarrow & (M;\Theta;\Psi;e[^{\vec{\varphi},\,\vec{v}}/_{\vec{t},\,\vec{x}}])\\
(M;\Theta;\Psi;\mathsf{let}\ x=\mathsf{get}\ \nu.n\ \mathsf{in}\ e) & \Longrightarrow & (M;\Theta;\Psi;e[^{M(\nu.n)}/_{x}])\\
\begin{array}{l}(M;\Theta,\nu\mapsto(\varphi,n);\Psi\\ \quad;\mathsf{let}\ x=\mathsf{put}[\nu,\tau]\ v\ \mathsf{in}\ e)\end{array} & \Longrightarrow & \begin{array}{l}(M,\nu.n\mapsto v;\Theta,\nu\mapsto(\varphi,n{+}1)\\ \quad;\Psi,\nu.n\mapsto\tau;e[^{\nu.n}/_{x}])\end{array}\\
(M;\Theta,\nu\mapsto(\varphi,n);\Psi;\mathsf{freergn}\ \nu;e) & \Longrightarrow & (M\backslash\nu;\Theta;\Psi\backslash\nu;e)\\
\begin{array}{l}(M;\Theta;\Psi;\mathsf{let}\ r=\mathsf{newrgn}\ \varphi\ \mathsf{in}\ e)\\ \quad\text{where } \bullet\vdash^{CiC}\varphi:\mathsf{Nat}\to\kappa\to\Omega\end{array} & \Longrightarrow & \begin{array}{l}(M;\Theta,\nu\mapsto(\varphi,0);\Psi;e[^{\nu}/_{r}])\\ \quad\text{where } \nu \text{ is fresh and } \bullet\vdash^{CiC}\nu:\mathsf{R}\ \kappa\end{array}\\
(M;\Theta,\nu\mapsto(\varphi',n);\Psi;\mathsf{cast}[P]\ \nu\mapsto\varphi;e) & \Longrightarrow & (M;\Theta,\nu\mapsto(\varphi,n);\Psi;e)\\
(M;\Theta,\nu\mapsto(\varphi',n');\Psi;\mathsf{set}\ \nu.n\mathrel{:\overset{\varphi}{=}}v;e) & \Longrightarrow & (M,\nu.n\mapsto v;\Theta,\nu\mapsto(upd\ \varphi'\ n\ \varphi,n');\Psi;e)\\
(M;\Theta;\Psi;\mathsf{let}\ x=\mathsf{cast}[P]\ v\ \mathsf{in}\ e) & \Longrightarrow & (M;\Theta;\Psi;e[^{v}/_{x}])\\
(M;\Theta;\Psi;\mathsf{let}\ \langle t,x\rangle=\mathsf{open}\ \langle\varphi,v\rangle\ \mathsf{in}\ e) & \Longrightarrow & (M;\Theta;\Psi;e[^{\varphi,\,v}/_{t,\,x}])\\
(M;\Theta;\Psi;\mathsf{let}\ \langle r,x\rangle=\mathsf{openrgn}\ \langle\rho,v\rangle\ \mathsf{in}\ e) & \Longrightarrow & (M;\Theta;\Psi;e[^{\rho,\,v}/_{r,\,x}])
\end{array}
$$

$$
\begin{array}{l}(M;\Theta;\Psi\\ \quad;\mathsf{iffirst}\ \nu\ (x,t\ .\ e_1)\ (e_2))\end{array} \Longrightarrow \begin{cases}(M;\Theta;\Psi;e_2) & \text{if } \Theta(\nu)=(\varphi,0)\\ (M;\Theta;\Psi;e_1[^{\tau,\,\nu.0}/_{t,\,x}]) & \text{where } \Psi(\nu.0)=\tau\end{cases}
$$

$$
\begin{array}{l}(M;\Theta;\Psi\\ \quad;\mathsf{ifnext}\ \nu.n\ (x,t\ .\ e_1)\ (e_2))\end{array} \Longrightarrow \begin{cases}(M;\Theta;\Psi;e_2) & \text{if } \Theta(\nu)=(\varphi,n{+}1)\\ (M;\Theta;\Psi;e_1[^{\tau,\,\nu.n+1}/_{t,\,x}]) & \text{where } \Psi(\nu.n{+}1)=\tau\end{cases}
$$

Figure 5.8: Operational semantics of the full language.

non-empty, x is bound to a pointer to the first object and t to its intended type.

$\mathsf{ifnext}\ v\ (x,t\ .\ e)\ (e)$: similarly, give a pointer to the next object past v, if any.

We also have existential packages to hide all those extra type details that we do not always want to have to deal with, as seen before for simple regions:

$\langle\varphi,v\rangle:\exists t\!:\!\kappa.\sigma$: create an existential package of v hiding the witness φ under name t.

$\mathsf{let}\ \langle t,x\rangle=\mathsf{open}\ v\ \mathsf{in}\ e$: unpack the existential v into t and x.

Also in some cases it can be difficult to keep track of the region in which lives an object, whereas it is easy to know which set of regions it lives in. So we have region aliases that allow us to give a new name to a region and indicate which set of real regions this can be an alias for. This is used extensively in our generational GC since the mutator generally does not know whether an object lives in the nursery or in the old space:

$\langle\rho, v\rangle : \exists r \in \vec{\rho}.\tau$ at $r.n$: create an existential region package which hides the region ρ under region alias r which can range over regions $\vec{\rho}$.

let $\langle r, x\rangle =$ openrgn v in e: unpack the existential region package v into variable x and region alias r.

To support such operations, we extend the definition of Θ slightly: the region spec stored in the region environment Θ is now either the type of the region together with its size (φ, n) or the set of regions for which this region can be an alias. The typing rules have to be updated also. The generational GC also wants to be able to extend the bound $\vec{\rho}$ of an existential region package, so we define a subsumption relation (defined as a CiC inductive type) $\sigma \triangleleft \sigma$ shown later in the typing rules. And we introduce a new operation cast$[P]$ v which returns value v but with its type changed to a supertype according to proof P.

5.5.1 Semantics

The operational semantics of the language are shown in Fig. 5.8. The semantics of the terms of the core language is left unchanged. The reduction rule for cast$[P]$ v is a no-op as expected. The scanning operations iffirst and ifnext both use Ψ to get the intended type of the new location. This of course would disappear after type-erasure.

The environment formation rules stay the same as before except for the following additional rule which specifies that a region alias is valid in Θ as long as the regions's kinds match:

$$\frac{\Delta \vdash \Theta \qquad \Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa \qquad \Delta \vdash^{CiC} \rho_i : \mathsf{R}\ \kappa}{\Delta \vdash \Theta, \rho \mapsto \vec{\rho}}$$

Note that regions can be aliases to other region aliases. They can also alias to dead regions. This allows freergn to free a region without caring about possible aliases. Of course the typing rules make sure such dangling aliases are themselves considered dead.

$\Delta \vdash \vec{\sigma} : \overrightarrow{t{:}\kappa}$	actual type arguments $\vec{\sigma}$ match formal arguments $\overrightarrow{t{:}\kappa}$
$\Theta \vdash \rho \in \vec{\rho}$	region ρ is a member of region set $\vec{\rho}$
$\Theta \lhd \Theta'$	region type environment Θ can be used where Θ' is expected
$\Delta \vdash \sigma \lhd \sigma'$	type σ is a subtype of σ'

$$\frac{\Delta \vdash^{CiC} \sigma : \kappa \qquad \Delta \vdash \vec{\sigma} : \overrightarrow{t{:}\kappa}[^{\sigma}/_{t}]}{\Delta \vdash \sigma, \vec{\sigma} : t{:}\kappa, \overrightarrow{t{:}\kappa}} \qquad \frac{\rho \in \vec{\rho}}{\Theta \vdash \rho \in \vec{\rho}} \qquad \frac{\Theta \vdash \rho_i \in \vec{\rho}'}{\Theta, \rho{\mapsto}\vec{\rho} \vdash \rho \in \vec{\rho}'}$$

$$\frac{}{\bullet \lhd \bullet} \qquad \frac{\Theta \lhd \Theta'}{\Theta, \rho{\mapsto}(\varphi, n) \lhd \Theta', \rho{\mapsto}(\varphi, n)} \qquad \frac{\Theta \lhd \Theta'}{\Theta, \rho{\mapsto}\vec{\rho} \lhd \Theta'} \qquad \frac{\Theta \lhd \Theta' \qquad \Theta \vdash \rho \in \vec{\rho}}{\Theta \lhd \Theta', \rho{\mapsto}\vec{\rho}}$$

$$\frac{}{\Delta \vdash \sigma \lhd \sigma} \qquad \frac{\Delta \vdash \sigma_i \lhd \sigma_i'}{\Delta \vdash \sigma_1 \times \sigma_2 \lhd \sigma_1' \times \sigma_2'} \qquad \frac{\begin{array}{c}\Delta \vdash^{CiC} \rho_i : \mathsf{R}\ \kappa \qquad \Delta \vdash^{CiC} \rho_j' : \mathsf{R}\ \kappa \\ \vec{\rho} \subset \vec{\rho}' \qquad \Delta, r{:}\mathsf{R}\ \kappa \vdash \sigma \lhd \sigma'\end{array}}{\Delta \vdash \exists r{\in}\vec{\rho}.\sigma \lhd \exists r{\in}\vec{\rho}'.\sigma'}$$

$\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma$	value v has type σ

$$\frac{}{\Psi; \Delta; \Theta; \Gamma \vdash n : \mathsf{int}} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v_i : \sigma_i}{\Psi; \Delta; \Theta; \Gamma \vdash (v_1, v_2) : \sigma_1 \times \sigma_2} \qquad \frac{}{\Psi; \Delta; \Theta; \Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Theta(\nu) = (\varphi, m) \Rightarrow \tau = \Psi(\nu.n)}{\Psi; \Delta; \Theta; \Gamma \vdash \nu.n : \tau\ \mathsf{at}\ \nu.n} \qquad \frac{\vdash \Delta \qquad \Delta \vdash \Theta \qquad \Delta \vdash \overrightarrow{x{:}\sigma} \qquad \Psi; \Delta; \Theta; \overrightarrow{x{:}\sigma} \vdash e}{\Psi; \Delta'; \Theta'; \Gamma \vdash \lambda[\Delta]\{\Theta\}(\overrightarrow{x{:}\sigma}).e : \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0}$$

$$\frac{\Delta \vdash^{CiC} \varphi : \kappa \qquad \Psi; \Delta; \Theta; \Gamma \vdash v : \sigma[^{\varphi}/_{t}]}{\Psi; \Delta; \Theta; \Gamma \vdash \langle \varphi, v \rangle : \exists t{:}\kappa.\sigma} \qquad \frac{\Theta \vdash \rho \in \vec{\rho} \qquad \Psi; \Delta; \Theta; \Gamma \vdash v : \sigma[^{\rho}/_{r}]}{\Psi; \Delta; \Theta; \Gamma \vdash \langle \rho, v \rangle : \exists r{\in}\vec{\rho}.\sigma}$$

$\Theta \vdash \tau\ \mathsf{at}\ \rho.n \mapsto \sigma$	pointer of type τ at $\rho.n$ points to an object of type σ
$\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \sigma$	value v points to an object of type σ
$\Psi; \Delta; \Theta; \Gamma \vdash op : \sigma$	operation op returns a value of type σ

$$\frac{}{\Theta, \rho{\mapsto}(\varphi, m) \vdash \tau\ \mathsf{at}\ \rho.n \mapsto \varphi\ n\ \tau} \qquad \frac{\Theta \vdash \rho \in \vec{\rho} \qquad \Theta \vdash \tau\ \mathsf{at}\ \rho_i.n \mapsto \varphi\ \rho_i}{\Theta \vdash \tau\ \mathsf{at}\ \rho.n \mapsto \varphi\ \rho}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \tau\ \mathsf{at}\ \rho.n \qquad \Theta \vdash \tau\ \mathsf{at}\ \rho.n \mapsto \sigma}{\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \sigma} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \sigma}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{get}\ v : \sigma}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma_1 \times \sigma_2}{\Psi; \Delta; \Theta; \Gamma \vdash \pi_i v : \sigma_i} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma' \qquad \Delta \vdash^{CiC} P : \sigma' \lhd \sigma}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{cast}[P]\ v : \sigma}$$

Figure 5.9: Static semantics of the full language: auxiliary judgments.

$$\boxed{\Psi;\Delta;\Theta;\Gamma \vdash e \quad \Big| \quad \text{expression } e \text{ is well-formed}}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash \mathit{op} : \sigma \qquad \Psi;\Delta;\Theta;\Gamma, x\!:\!\sigma \vdash e}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ x = \mathit{op}\ \mathsf{in}\ e} \qquad \frac{\Psi;\Delta;\bullet;\Gamma \vdash v : \mathsf{int}}{\Psi;\Delta;\bullet;\Gamma \vdash \mathsf{halt}\ v}$$

$$\frac{\begin{array}{c}\Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa \qquad \Delta \vdash^{CiC} \tau : \kappa \qquad \Psi;\Delta;\Theta;\Gamma \vdash v : \varphi\ n\ \tau \\ \Psi;\Delta;\Theta, \rho \mapsto (\varphi, n{+}1);\Gamma, x\!:\!\tau\ \mathsf{at}\ \rho.n \vdash e\end{array}}{\Psi;\Delta;\Theta, \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{let}\ x = \mathsf{put}[\rho,\tau]\ v\ \mathsf{in}\ e}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Theta;\Gamma \vdash v \mapsto \forall[\overrightarrow{t\!:\!\kappa}]\{\Theta'\}(\vec{\sigma}) \to 0 \\ \Delta \vdash \vec{\varphi} : \overrightarrow{t\!:\!\kappa} \quad \Theta \triangleleft \Theta'[\vec{\varphi}/\vec{t}\,] \quad \Psi;\Delta;\Theta;\Gamma \vdash v_i : \sigma_i[\vec{\varphi}/\vec{t}\,]\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash v[\vec{\varphi}](\vec{v})} \qquad \frac{\begin{array}{c}\Delta \vdash^{CiC} \varphi : \mathsf{Nat} \to \kappa \to \Omega \\ \Psi;\Delta, r\!:\!\mathsf{R}\ \kappa;\Theta, r \mapsto (\varphi, 0);\Gamma \vdash e\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e}$$

$$\frac{\begin{array}{c}\Delta \vdash^{CiC} P : \Pi i\!:\!0..n{-}1.\,\Pi t\!:\!\kappa.\,(\varphi\ i\ t) \triangleleft (\varphi'\ i\ t) \\ \Psi;\Delta;\Theta, \rho \mapsto (\varphi', n);\Gamma \vdash e \qquad \Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa\end{array}}{\Psi;\Delta;\Theta, \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{cast}[P]\ \rho \mapsto \varphi'; e} \qquad \frac{\Psi;\Delta;\Theta;\Gamma \vdash e}{\Psi;\Delta;\Theta, \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{freergn}\ \rho; e}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Theta, \rho \mapsto (\varphi, n);\Gamma \vdash v : \tau\ \mathsf{at}\ \rho.m \\ \Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa \qquad \Delta \vdash^{CiC} \varphi' : \kappa \to \Omega \qquad \Psi;\Delta;\Theta, \rho \mapsto (\varphi, n);\Gamma \vdash v' : \varphi'\ \tau \\ \Psi;\Delta;\Theta, \rho \mapsto (\mathit{upd}\ \varphi\ m\ \varphi', n);\Gamma \vdash e\end{array}}{\Psi;\Delta;\Theta, \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{set}\ v \overset{\varphi'}{:=} v'; e}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Theta;\Gamma \vdash v : \exists t'\!:\!\kappa.\sigma \\ \Psi;\Delta, t\!:\!\kappa;\Theta;\Gamma, x\!:\!\sigma[t/t'] \vdash e\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ \langle t, x\rangle = \mathsf{open}\ v\ \mathsf{in}\ e} \qquad \frac{\begin{array}{c}\Psi;\Delta;\Theta;\Gamma \vdash v : \exists r' \in \vec{\rho}.\sigma \qquad \Delta \vdash^{CiC} \rho_i : \mathsf{R}\ \kappa \\ \Psi;\Delta, r\!:\!\mathsf{R}\ \kappa;\Theta, r \mapsto \vec{\rho};\Gamma, x\!:\!\sigma[r/r'] \vdash e\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ \langle r, x\rangle = \mathsf{openrgn}\ v\ \mathsf{in}\ e}$$

$$\frac{\begin{array}{c}\Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa \qquad \Psi;\Delta;\Theta, \rho \mapsto (\varphi, 0);\Gamma \vdash e_2 \\ \Psi;\Delta, t\!:\!\kappa;\Theta, \rho \mapsto (\varphi, n);\Gamma, x\!:\!t\ \mathsf{at}\ \rho.0 \vdash e_1\end{array}}{\Psi;\Delta;\Theta, \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{iffirst}\ \rho\ (x, t\ .\ e_1)\ (e_2)}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Theta;\Gamma \vdash v : \tau\ \mathsf{at}\ \rho.m \qquad \Psi;\Delta;\Theta, \rho \mapsto (\varphi, m{+}1);\Gamma \vdash e_2 \\ \Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa \qquad \Psi;\Delta, t\!:\!\kappa;\Theta, \rho \mapsto (\varphi, n);\Gamma, x\!:\!t\ \mathsf{at}\ \rho.m{+}1 \vdash e_1\end{array}}{\Psi;\Delta;\Theta, \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{ifnext}\ v\ (x, t\ .\ e_1)\ (e_2)}$$

Figure 5.10: Static semantics of the full language: well-formed terms.

Figure 5.9 shows the formation rules for values, operations and auxiliary judgments. Large parts are kept exactly as they were for the core language, so I will concentrate on the differences. The new judgment $\Theta \vdash \rho \in \vec{\rho}$ checks whether ρ is known to be one of $\vec{\rho}$, given the aliasing info provided in Θ. This is used to check the validity of new region aliases. The equivalence relation $\Theta \sim \Theta'$ is replaced by $\Theta \lhd \Theta'$. This new relation checks that if the current environment is Θ, we can call a function that expects Θ', which is true if the real regions are equal and if the call drops some region aliases, or if it creates some new region aliases whose validity we can infer from Θ. The formation rule for $\nu.n$ is slightly modified to check that ν is a real live region and not a dead alias.

The auxiliary rule $v \mapsto \sigma$ is extended, with the help of another auxiliary rule τ at $\rho.n \mapsto \sigma$, to handle region aliases. If ρ is a region alias, we could simply require that the actual type be the same, independently from the region this is an alias for, but that would be too restrictive. Instead, we put no restriction of any sort and simply allow the type σ to depend on ρ. For example, if $\Theta \vdash \rho \in \rho_1, \rho_2$ and the actual type in ρ_1 is σ_1 whereas the actual type in ρ_2 is σ_2, the actual type in ρ can be "if $(\rho = \rho_1)$ (σ_1) (σ_2)". More often, σ_1 and σ_2 share some structure, as in "$\mathsf{int} \times \tau$ at $\rho_1.n$" and "$\mathsf{int} \times \tau$ at $\rho_2.n$" where the actual type in ρ can naturally be "$\mathsf{int} \times \tau$ at $\rho.n$". Inferring such types is not possible in general, so type-checking might need to impose additional constraints or to require additional type annotations in order to be decidable.

Figure 5.10 shows the formation rules for the main terms. In the terms taken from the core language, the only changes are that function calls now use $\Theta \lhd \Theta'$ instead of $\Theta \sim \Theta'$ and that cast uses $\sigma \lhd \sigma'$ instead of equality. Most importantly, set is unchanged and does not allow assignment through a region alias, since it would not know which region's type to update. We might thus need to extend the language again with a new set term that allows assignment through a region alias but requires the old and new actual types to match so that the region's type does not need to be updated.

5.5.2 Properties of the full language

The full language enjoys the same type preservation and progress properties as the core language. These are proved in A.2.2. It also enjoys the complete collection property, whose proof is the same as it was for the core calculus.

5.6 Generational collection

The simple type-preserving GCs developed until now have a few important limitations. The most important one is that cyclic garbage is not properly collected. Another limitation is that generational collectors do not allow the mutator (i.e. the code that uses the GC) to create references from the old region to the young via destructive assignment. I will show in this section how to use typed regions to lift those restrictions that make existing type-preserving GCs impractical.

5.6.1 Simple type-preserving GC

As seen in the previous chapter, the basic idea of a type-preserving GC, proposed by Wang and Appel [WA01], is to layer a stop&copy collector on top of a region calculus, where the whole heap is placed in a single region and where the *copy* function copies the heap from the *from* region to a new *to* region and then frees the *from* region.

Because the type of the heap contains region annotations which will necessarily be different before copying than after, the *copy* function cannot just be of type $t \to t$ but instead has to be of the form $M_F(t) \to M_T(t)$ where t represents what should be preserved while the M type function annotates it with details that the mutator does not care about. Furthermore, in order to work correctly, the *copy* routine might require things like tag bits, mark bits or in case of generational GC it will need to make sure the mutator obeyed the generation barriers and provides a correct remembered set. All those added constraints will need to be somehow encoded in M since *copy* has obviously no control over t.

Note that M is really a function of two arguments. I write $M_F(t)$ instead of $M\ F\ t$ only to make the special role of t more visible and to parallel the notation used in the previous chapter.

A good way to look at it is that t is a high-level, GC-oblivious type of the heap, while $M_F(t)$ is the low-level representation with all the added details necessary for the correct functioning of the GC. The two will generally not be of the same kind; t might for example correspond to the type used in the high-level source language of the mutator code.

The simplest case is a bare-bones stop&copy GC with a source language that only supports integers and pairs. M might then look like the following, where R is the current region holding the heap:

$$
\begin{array}{lcl}
M_R(\mathsf{int}) & \Rightarrow & \mathsf{int} \\
M_R(\tau_1 \times \tau_2) & \Rightarrow & (M_R(\tau_1) \times M_R(\tau_2)) \text{ at } R
\end{array}
$$

In the case of a 2-generation collector, the heap will be spread over 2 regions: Y for the nursery and O for the old generation. M will now have to enforce that there cannot be any pointer back from O to Y by only allowing $(M_{O,Y}(\tau_1) \times M_{O,Y}(\tau_2))$ at Y and $(M_{O,O}(\tau_1) \times M_{O,O}(\tau_2))$ at O:

$$
\begin{array}{lcl}
M_{O,A}(\mathsf{int}) & \Rightarrow & \mathsf{int} \\
M_{O,A}(\tau_1 \times \tau_2) & \Rightarrow & \exists r \in \{O, A\}.(M_{O,r}(\tau_1) \times M_{O,r}(\tau_2)) \text{ at } r
\end{array}
$$

In this definition, A is expected to be either equal to O or Y depending on whether pointers to Y are allowed at this place.

5.6.2 Generational GC

We want to create a simple yet usable type-preserving generational GC. In order to be usable, it needs to be able to properly handle cycles in the heap and it needs to allow the mutator to use destructive assignment, even if that means creating pointers back from the old generation to the nursery. The traditional way to deal with such pointers is to keep track of all of them in a *remembered-set*, but for simplicity, we will restrict mutability to ref-cells, as is done in SML, and place all those ref-cells in a separate region which will thus play the role of a degenerate remembered-set.

Compared to the situations outlined in the previous section, the heap is now spread over three regions: Y for the nursery, O for the old generation and R for the ref-cells. So that the collection of Y can take place without having to scan O, we need to make sure that there is no pointer from O to Y. I.e. pointers from Y and R can point to any of Y, O or R but pointers from O can only point to O or R. Our M type function which maps high-level types to their low-level representation will again have to enforce this constraint. If our high-level types include integers, ref-cells and pairs, M could be defined as follows:

$$
\begin{array}{lcl}
M_{Y,R,O,A}(\mathsf{int}) & \Rightarrow & \mathsf{int} \\
M_{Y,R,O,A}(\mathsf{ref}\ \tau) & \Rightarrow & M_{Y,R,O,Y}(\tau)\ \mathsf{at}\ R \\
M_{Y,R,O,A}(\tau_1 \times \tau_2) & \Rightarrow & \exists r \in \{O, A\}.(M_{Y,R,O,r}(\tau_1) \times M_{Y,R,O,r}(\tau_2))\ \mathsf{at}\ r
\end{array}
$$

Here A is expected to be either equal to O or to Y and the type $M_{Y,R,O,A}(\tau_1 \times \tau_2)$ is the type of a pair that can be allocated either in O or in A and whose children (and grand-children, etc...) might also refer additionally to R or to Y.

The GC code that copies objects from the nursery Y to the old space O takes the type of the heap t, the regions Y, O, and R, and the heap itself $h : M_{Y,R,O,Y}(t)$ as well as a continuation k. It first copies everything reachable from the root h (but only within Y) to O. Then scans all the ref-cells, considered as extra roots, and copies anything reachable from them as well. Finally, it frees Y and creates a new nursery before returning to the continuation k. It could look like the following:

$$\begin{aligned}&\boldsymbol{GC}[t\!:\!\Omega^\tau]\{Y,R,O\}(h\!:\!M_{Y,R,O,Y}(t),k\!:\!\dots)\\&= \mathsf{let}\ h = \mathit{copy}[t,Y,R,O](h)\ \mathsf{in}\\&\quad \mathsf{let}\ t',x = \mathsf{first}\ R\ \mathsf{in}\\&\quad \mathsf{let}\ \_ = \mathit{redirect}[t',Y,R,O](x)\ \mathsf{in}\\&\quad \mathsf{freergn}\ Y; \mathsf{let}\ Y = \mathsf{newrgn}\ \ \mathsf{in}\ k[Y,R,O](h)\end{aligned}$$

where the *copy* function should simply do a generic deep copy, but only within the bounds of Y and the *redirect* function should scan the R region, redirecting any pointer to the Y region to a copy of that object in O. The type of the new root pointer h returned by *copy* can be described as $M_{Y,R,O,O}(t)$.

In order to be able to free Y at the end of the collection, we need to keep track somewhere at the type-level of the fact that pointers in R that have been redirected can only point to O or to R but not to Y any more. Indeed, as the redirection is taking place, the type of h keeps changing gradually until the end when it becomes equivalent to $M_{O,R,O,O}(t)$ and thus independent from Y. If b is the boundary between what has been redirected and what has not, then we can write $h\!:\!C_{Y,R,O,O,b}(t)$ where C is defined as:

$$\begin{array}{lcl}C_{Y,R,O,A,b}(\mathsf{int}) & \Rightarrow & \mathsf{int}\\ C_{Y,R,O,A,b}(\mathsf{ref}\ \tau) & \Rightarrow & \mathsf{let}\ r = \mathsf{if}\ (\mathit{before}\ b)\ (O)\ (Y)\ \mathsf{in}\ C_{Y,R,O,r,b}(\tau)\ \mathsf{at}\ R\\ C_{Y,R,O,A,b}(\tau_1\times\tau_2) & \Rightarrow & \exists r\in\{O,A\}.(C_{Y,R,O,r,b}(\tau_1)\times C_{Y,R,O,r,b}(\tau_2))\ \mathsf{at}\ r\end{array}$$

This is almost the same as the previous M, except for the added b parameter and the case of ref where the object reachable from the ref-cell will have either type $C_{Y,R,O,O,b}(\tau)$ or $C_{Y,R,O,Y,b}(\tau)$ depending on whether the cell has already been redirected or not.

The above presentation is sloppy and simplistic but already requires unusual features from the language:

- The *before* b test in the definition of $C_{Y,R,O,A,b}(\mathsf{ref}\ \tau)$ obviously needs to know not just the region in which the object is located but its actual location, which implies that object locations need to be reflected in the pointer type.

- The new operation first (and its companion ifnext which will be needed inside *redirect*), that allows the code to scan the region R, should return both a new pointer x and a new type t' where x is expected to be of type $C_{Y,R,O,Y,b}(t')$. But this is only safe if we know that all objects in the relevant section have a type of the form $C_{Y,R,O,Y,b}(\tau)$.

- Since the type of h, indexed by the boundary b keeps changing as the boundary moves, we need to be able to update its type when the assignment $x := \mathit{copy}[..](..)$ moves the boundary, even though we do not know anything about the aliasing relationship between x and h.

Those same three issues appear when trying to write a Cheney-style stop&copy GC where the *to* region is used as a queue and needs to be scanned and redirected in very similar ways.

5.6.3 Solution

Now, using typed regions, the above problems become very easy to solve. int, $\mathsf{ref}\ \tau$, and $\tau_1 \times \tau_2$ are defined as a CiC inductive type of kind Ω^τ. Region R has kind $\mathsf{R}\ \Omega^\tau$, regions Y and O have kind $\mathsf{R}\ (\Omega^\tau \times \Omega^\tau)$. The type of our variables have the shape $M_{O,R,Y}(\tau)$ defined below:

$$\begin{aligned}
M_{O,R,A}(\mathsf{int}) &\Rightarrow \mathsf{int}\\
M_{O,R,A}(\mathsf{ref}\ \tau) &\Rightarrow \exists n\!:\!\mathsf{Nat}.\tau\ \mathsf{at}\ R.n\\
M_{O,R,A}(\tau_1 \times \tau_2) &\Rightarrow \exists r\in\{O,A\}.\exists n\!:\!\mathsf{Nat}.(\tau_1,\tau_2)\ \mathsf{at}\ r.n
\end{aligned}$$

where (τ_1, τ_2) is a pair of kind $\Omega^\tau \times \Omega^\tau$. The regions's types while the mutator is running are:

$$\begin{aligned}
Y &\mapsto \lambda n.\lambda(t_1,t_2).M_{O,R,Y}(t_1) \times M_{O,R,Y}(t_2)\\
O &\mapsto \lambda n.\lambda(t_1,t_2).M_{O,R,O}(t_1) \times M_{O,R,O}(t_2)\\
R &\mapsto \lambda n.\lambda t.M_{O,R,Y}(t)
\end{aligned}$$

(* Definition of the source-level types *)
Inductive Ω^τ : Kind $= \mathit{int} : \Omega^\tau \mid \mathit{ref} : \Omega^\tau \to \Omega^\tau \mid \mathit{pair} : \Omega^\tau \to \Omega^\tau \to \Omega^\tau$

(* Mapping types from source to low-level *)
typedef $\boldsymbol{M}_{o,r,a}(t)$
$\quad = \mathsf{case}\ t\ \mathsf{of}\ \mathit{int} \Rightarrow \mathsf{int} \mid\ \mathit{ref}\ t \Rightarrow \exists n.t\ \mathsf{at}\ r.n \mid\ \mathit{pair}\ t_1\ t_2 \Rightarrow \exists b\in\{o,a\}.\exists n.(t_1,t_2)\ \mathsf{at}\ b.n$

(* Type of a region holding tuples *)
typedef $\boldsymbol{TupRgn}\ o\ r\ a = \lambda n.\lambda(t_1,t_2).M_{o,r,a}(t_1) \times M_{o,r,a}(t_2)$

(* Type of a region holding ref cells *)
(* b: scan boundary during redirection *)
typedef $\boldsymbol{RefRgn}\ o\ r\ y\ b = \lambda n.\lambda t.\mathsf{let}\ a = \mathsf{if}\ (b > n)\ o\ y\ \mathsf{in}\ M_{o,r,a}(t)$

(* Type of the continuation passed to *GC* *)
typedef $\boldsymbol{gccont} = \forall[y{:}\mathsf{R}\ (\Omega^\tau \times \Omega^\tau)]\{\ldots\}(x{:}M_{o,r,y}(t)) \to 0$

(* A deep copy function not shown here *)
$\boldsymbol{copy} : \forall[t{:}\Omega^\tau, b{:}\mathsf{Nat},\ldots]\{\ldots\}(M_{o,r,y}(t)) \to M_{o,r,o}(t)$

(* Main entry point *)
(* h is the root pointer and t its type *)
$\boldsymbol{GC}[t{:}\Omega^\tau, y, o{:}\mathsf{R}\ (\Omega^\tau \times \Omega^\tau), r : \mathsf{R}\ \Omega^\tau, \ldots]$
$\quad \{y \mapsto (\mathit{TupRgn}\ o\ r\ y, m_y), o \mapsto (\mathit{TupRgn}\ o\ r\ o, m_o), r \mapsto (\lambda n.\lambda t.M_{o,r,y}(t), m_r)\}$
$\quad (h{:}M_{o,r,y}(t), k{:}\mathit{gccont})$
$= \mathsf{let}\ h = \mathit{copy}[t,0,y,r,o](h)\ \mathsf{in}$
$\quad \mathsf{iffirst}\ r\ \mathsf{then}\ x,t'\ .\ \mathit{redirect}[t,0,t',y,r,o](x,h,k)$
$\quad \mathsf{else}\ \mathit{GCtail}[t,y,r,o](h,k)$

(* Finish up GC when redirection is done *)
$\boldsymbol{GCtail}[t{:}\Omega^\tau, y, o{:}\mathsf{R}\ (\Omega^\tau \times \Omega^\tau), r : \mathsf{R}\ \Omega^\tau, \ldots]$
$\quad \{y \mapsto (\mathit{TupRgn}\ o\ r\ y, m_y), o \mapsto (\mathit{TupRgn}\ o\ r\ o, m_o), r \mapsto (\mathit{RefRgn}\ o\ r\ y\ m_r, m_r),\}$
$\quad (h{:}M_{o,r,o}(t), k{:}\mathit{gccont})$
$= \mathsf{freergn}\ y;$
$\quad \mathsf{let}\ y = \mathsf{newrgn}\ (\lambda\_{:}(\Omega^\tau \times \Omega^\tau).\,\mathsf{int})\ \mathsf{in}$
$\quad \mathsf{cast}[\ldots]\ r \mapsto (\lambda n.\lambda t.M_{o,r,y}(t));$
$\quad \mathsf{cast}[\ldots]\ y \mapsto (\mathit{TupRgn}\ o\ r\ y);$
$\quad k[y](\mathsf{cast}[\ldots : M_{o,r,o}(t) \lhd M_{o,r,y}(t)]\ h)$

(* Scan r, redirecting ptrs from y to o *)
(* h: root pointer to pass back to *GCtail* *)
(* x: boundary pointer that sweeps r *)
$\boldsymbol{redirect}[t{:}\Omega^\tau, b{:}\mathsf{Nat}, t'{:}\Omega^\tau, y, o{:}\mathsf{R}\ (\Omega^\tau \times \Omega^\tau), r : \mathsf{R}\ \Omega^\tau, \ldots]$
$\quad \{y \mapsto (\mathit{TupRgn}\ o\ r\ y, m_y), o \mapsto (\mathit{TupRgn}\ o\ r\ o, m_o), r \mapsto (\mathit{RefRgn}\ o\ r\ y\ b, m_r)\}$
$\quad (x{:}t\ \mathsf{at}\ r.b, h{:}M_{o,r,o}(t'), k{:}\mathit{gccont})$
$= \mathsf{set}\ x \overset{M_{o,r,o}}{:=} \mathit{copy}[t,b,y,r,o](\langle y, \mathsf{get}\ x\rangle);$
$\quad \mathsf{cast}[\ldots]\ r \mapsto (\mathit{RefRgn}\ o\ r\ y\ (b{+}1));$
$\quad \mathsf{ifnext}\ x\ \mathsf{then}\ x,t\ .\ \mathit{redirect}[t,b{+}1,t',y,r,o](x,h,k)$
$\quad \mathsf{else}\ \mathit{GCtail}[t,y,r,o](h)$

Figure 5.11: Generational GC

where $\lambda(t_1, t_2)$ is just a pattern matching short hand. While a collection copying objects from Y to O is in progress and pointers in R are redirected from Y to O, the type of Y and O stay the same, and the same M function is still used. Only R needs to be changed to the following:

$$R \mapsto \lambda n.\lambda t.\mathsf{let}\ r = \mathsf{if}\ (m > n)\ (O)\ (Y)\ \mathsf{in}\ M_{O,R,r}(t)$$

The top-level code of the GC code that copies objects from the nursery Y to the old space O is shown in figure 5.11. We omit proof arguments as well as some type parameters and use direct-style rather than continuation passing style where convenient, to make the code more readable. The code of the *copy* function is not shown. A *copy* function similar to the one presented in last chapter could be used.

Since the types of regions R, O, and Y all refer to each other, there is a problem of circularity that prevents us from creating a region with its proper type. In *GCtail* we show how to work around such problems using cast: we first create Y with a dummy type, and then cast it to the desired type (the proof needs to show that the new type preserves the type of all objects in the region which is trivial since the region is empty).

5.7 Related work

The calculus of capabilities [CWM99] was the first calculus to provide safe explicit memory deallocation while allowing dangling pointers. The linear handling of our regions was strongly influenced by that work.

In the work on TAL [MWCG98], the authors showed a simple way to handle the problem of separating allocation from object initialization, without resorting to any form of linearity.

Alias types [SWM00, WM00] was also a major source of inspiration for our system. It's a type system designed specifically to handle low-level code. In that work, strong update is the only form of update available. Separating object initialization from allocation is very easy, as is explicit deallocation and memory reuse.

The Vault language [DF01] took the work on alias types and both extended it and gave it a surface syntax (so as to enable the programmer to give that needed aliasing information). In the first paper, they mostly showed how to integrate classical intuitionistic references with alias-types-style statically tracked references. They also showed that tracking references to region objects allows their system to subsume a region type system. The main difference compared to our work is that you have to choose once and for all whether a reference should be intuitionistic or linear (i.e. *tracked*).

In a subsequent paper [FD02] they addressed that limitation by introducing the operators *adoption*, which allows the user to make a linear reference intuitionistic, and *focus*, which does the converse. This significantly increases the expressiveness of their user-level language. Sadly, these constructs are not applicable to our GC problem for two reasons: First, *adoption* is too high-level for us and seems difficult to adapt to a low-level language. Second, although *focus* does allow some of what we need (such as strong update in the presence of arbitrary aliasing), it requires the type change to be transient, limited to the scope of the construct.

Wang and Appel [WA99] built a tracing garbage collector on top of a region-based calculus, thus providing both type safety and completely automatic memory management.

Vanderwaart and Crary [VC03] design a type system that enforces that the programs correctly manipulate the details of the stack layout required by a particular GC which is kept implicit. The system is designed for a sophisticated GC which uses a static table indexed by the return address to describe the activation frames, much like stack-walking implementations of exception handling. The type system checks among other things that the table provided by the program is correct and used consistently.

Shao et al. [SSTP02] proposed to use CiC as the type calculus of a programming language. This allows sophisticated type manipulation and enables programs to express arbitrary properties of the values manipulated. I reuse their idea with the same purpose but by virtue of the rest of the type system I can additionally capture arbitrary properties of the state.

5.8 Conclusion

I have presented a novel type system that offers an unusual flexibility to play with the typing of memory locations. This type system offers the ability to choose any mix of linear or intuitionistic typing of references and to change this choice over time to adapt it to the current needs. It is able to handle strong update of memory locations even in the presence of unknown aliasing patterns. The reliance on CiC allows very sophisticated type manipulations.

We have shown how to use it to solve the difficult problem of type checking a generational garbage collector, including proper handling of cycles and while allowing the mutator to perform destructive assignment on reference cells. We have also shown how to use this same type system to encode a stack of activation records inside a region.

Chapter 6

Future directions

I have shown that types can be used to formalize and reason about complex properties of code such as the behavior of a code inliner and the safety of a generational garbage collector.

The language I developed for the garbage collector is very powerful and opens up many new possibilities. It is also highly unconventional and represents an intriguing point on the design space of type systems. Many future directions seem possible.

6.1 Implementation

The most immediately needed development is an actual implementation of such a language, so as to get more experience with its power and limitations. Especially challenging will be the issue of managing the cost of the heavy type annotations and the extensive use of complex operations at the type level which will have to be interpreted throughout the compilation.

Another issue with an implementation is to make the language more realistic. This mostly involves replacing put and get with alloc, load, and store which is easy enough but will also require splitting the region handles into a run-time and a compile-time component. This should not present any particular theoretical difficulty.

Finally, any reasonable implementation will need to work within the scope of a compiler for a high-level language and compiling the intermediate language into a language such as my typed regions will require substantial work. An important difficulty will be encoding recursive type in the inductive definition of Ω, for example.

6.2 Advanced uses

Another venue is to go further in the direction of encoding properties in the types. It should be easy to extend the system with singleton integers and use them to explicitly encode GC tags, forwarding pointer bits, and remembered sets. Together with a Cheney-style copy routine, it should be possible to type-check a production quality garbage collector.

Also, compiling such a typed regions system to proof carrying code could present substantial difficulties because of the power the type system I use. It seems for example impossible to prove the soundness of the language using a deep embedding into Coq.

6.3 Surface syntax

As it stands the system of typed regions can only be used by a dedicated individual with a very good understanding of sophisticated type systems.

But the way the type system is structured makes it possible to move a significant part of the complex invariants to a few specific places such as in the regions's types, leaving most of the code uncluttered as long as it sticks to the traditional usage patterns of a simple region system. With a bit of syntactic sugar and a few special shorthands, it should be possible to make it into a usable source language. How to keep the source language understandable while still making the full power of the underlying system available is probably going to be a problem.

As mentioned before, this system represents an unusual point in the design space of type systems. It seems very likely that yet different points in the design space will prove at least equally useful. There is a lot of room for experimentation in that area as well.

Bibliography

[AJ97] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, September 1997. Princeton TR-556-97.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[App91] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1991. ISBN 0-521-41695-7.

[App94] Andrew W. Appel. Loop headers in lambda-calculus or CPS. *Lisp and Symbolic Computation*, 7:337–343, June 1994. Princeton TR-460-94.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[Asa99] Kenichi Asai. Binding-time analysis for both static and dynamic expressions. In *Static Analysis Symposium*, pages 117–133, September 1999.

[Ash97] Michael J. Ashley. The effectiveness of flow analysis for inlining. In *International Conference on Functional Programming*, pages 99–111. ACM Press, June 1997.

[BA97] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *International Conference on Functional Programming*, pages 112–124. ACM Press, June 1997.

[Bar91] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science (volume 2)*. Oxford Univ. Press, 1991.

[BC99] Guy Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Symposium on Programming Languages Design and Implementation*, pages 104–107. ACM Press, May 1999.

[BKR98] Nick Benton, Andrew Kennedy, and George Russel. Compiling Standard ML to Java bytecodes. In *International Conference on Functional Programming*, pages 129–140. ACM Press, September 1998.

[Blu95] Matthias Blume. *Standard ML of New Jersey Compilation Manager*, 1995. Manual accompanying SML/NJ software.

[BW95] Lars Birkedal and Morten Welinder. Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, 8(3):191–208, September 1995.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CN96] Charles Consel and Francois Noël. A general approach for run-time specialization, and its application to C. In *Symposium on Principles of Programming Languages*, pages 145–156. ACM Press, January 1996.

[Con93] Charles Consel. Polyvariant binding-time analysis for applicative languages. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, 1993.

[Cra00] Karl Crary. Typed assembly language: Type theory for machine code. Talk presented at 2000 PCC Workshop, Santa Barbara, CA, June 2000.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *International Conference on Functional Programming*, pages 233–248. ACM Press, September 1999.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming*, pages 301–312. ACM Press, September 1998.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, TX, January 1999.

[Dan96] Olivier Danvy. Type-directed partial evaluation. In *Symposium on Principles of Programming Languages*, pages 242–257. ACM Press, January 1996.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *11th Annual Symposium on Logic in Computer Science*, pages 184–195, July 1996.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Symposium on Programming Languages Design and Implementation*. ACM Press, May 2001.

[DL92] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In *Conference on Lisp and Functional Programming*, pages 299–310, San Francisco, CA, 1992.

[DMP96] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Symposium on Principles of Programming Languages*, pages 258–270. ACM Press, January 1996.

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Symposium on Programming Languages Design and Implementation*. ACM Press, May 2002.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of qualified types. In *Symposium on Programming Languages Design and Implementation*, pages 192–203. ACM Press, May 1999.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Symposium on Programming Languages Design and Implementation*, pages 237–247, New York, June 1993. ACM Press.

[Gir72] J. Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, University of Paris VII, 1972.

[GSSS01] Kevin Glynn, Peter J. Stuckey, Martin Sulzmann, and Harald Sndergaard. Boolean constraints for binding-time analysis. In *Program as Data Objects*, pages 39–62, 2001.

[GST01] Steve Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In *International Conference on Functional Programming*. ACM Press, September 2001.

[HM94] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In *European Symposium on Programming*, pages 287–301, Edinburgh, UK, April 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Symposium on Principles of Programming Languages*, pages 130–141, January 1995.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Symposium on Principles of Programming Languages*, pages 341–344, New York, January 1990. ACM Press.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[How80] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[HPM+00] Gerard Huet, Christine Paulin-Mohring, et al. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1, May 2000.

[HST+02] Nadeem Abdul Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generations*. Prentice Hall International, 1993.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Symposium on Principles of Programming Languages*, pages 177–188, January 1992.

[Ler95] Xavier Leroy. Le système Caml Special Light: Modules et compilation efficace en Caml. Technical Report 2721, Institut National de Recherche en Informatique et Automatique, nov 1995.

[MBS99] Stefan Monnier, Matthias Blume, and Zhong Shao. Cross-functor inlining in FLINT. Technical Report YALEU/DCS/TR-1189, Dept. of Computer Science, Yale University, New Haven, CT, March 1999.

[MCB99] Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental run-time specialization for free. In *Symposium on Programming Languages Design and Implementation*, pages 281–292. ACM Press, May 1999.

[MG98] Yasuhiko Minamide and Jacques Garrigue. On the runtime complexity of type-directed unboxing. In *International Conference on Functional Programming*, pages 1–12. ACM Press, September 1998.

[MH97] Greg Morrisett and Robert Harper. A comprehensive treatment of typed closure conversion. In *Higher Order Operational Techniques in Semantics*, Stanford, CA, December 1997.

[MMH96] Yasuhiko Minimide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, January 1996.

[Mog88] Eugenio Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, 1988.

[Mog97] Eugenio Moggi. A categorical account of two-level languages. In *13^{th} Conference on the Mathematical Foundations of Programming Semantics*, March 1997.

[Mor00] Greg Morrisett. Open problems for certifying compilers. Talk presented at 2000 PCC Workshop, Santa Barbara, CA, June 2000.

[MTBS99] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: simpler, and more expressive. In *European Symposium on Programming*, pages 193–207, Amsterdam, The Netherlands, March 1999.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David B. MacQueen. *The Definition of Standard ML Revised*. MIT Press, Cambridge, Massachusetts, 1997.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Symposium on Principles of Programming Languages*, pages 85–97, San Diego, CA, January 1998.

[Nec97] George C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages*, January 1997.

[NL98] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *Symposium on Programming Languages Design and Implementation*, pages 333–344, New York, NY, June 1998. ACM Press.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages.* Cambridge University Press, New York, 1992.

[NO93] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *Symposium on Programming Languages Design and Implementation*, New York, June 1993. ACM Press.

[Pau93] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*. LNCS 664, Springer-Verlag, 1993.

[PM99] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. In *Proceedings of the International Workshop on Implementation of Declarative Languages*, Paris, France, September 1999.

[PMar] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, to appear.

[PS98] Simon Peyton Jones and André L M Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, pages 717–740, 1972.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

[SA94] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 150–161, Orlando, FL, June 1994.

[SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Symposium on Programming Languages Design and Implementation*, pages 116–129, La Jolla, CA, June 1995. ACM Press.

[Sha97a] Zhong Shao. Flexible representation analysis. In *International Conference on Functional Programming*, pages 85–98. ACM Press, June 1997.

[Sha97b] Zhong Shao. An overview of the FLINT/ML compiler. In *International Workshop on Types in Compilation*, June 1997.

[Sha98] Zhong Shao. Typed cross-module compilation. In *International Conference on Functional Programming*, pages 141–152. ACM Press, September 1998.

[Sha99] Zhong Shao. Transparent modules with fully syntactic signatures. In *International Conference on Functional Programming*, pages 220–232, Paris, France, September 1999. ACM Press.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

[SLM98] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *International Conference on Functional Programming*, pages 313–323. ACM Press, September 1998.

[SS98] Bratin Saha and Zhong Shao. Optimal type lifting. In *International Workshop on Types in Compilation*, March 1998.

[SSTP02] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Symposium on Principles of Programming Languages*, pages 217–232, Portland, OR, January 2002.

[ST96] Michael Sperber and Peter Thiemann. Realistic compilation by partial evaluation. In *Symposium on Programming Languages Design and Implementation*, pages 206–214, Philadelphia, PA, May 1996. ACM Press.

[ST97] Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *Symposium on Programming Languages Design and Implementation*, pages 215–225. ACM Press, June 1997.

[STS00] Bratin Saha, Valery Trifonov, and Zhong Shao. Fully reflexive intensional type analysis. Technical Report YALEU/DCS/TR-1194, Dept. of Computer Science, Yale University, New Haven, CT, March 2000.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, 2000.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute, Beaverton, Oregon, November 1999.

[Tar96] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, Carnegie Mellon University, December 1996. CMU-CS-97-108.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Symposium on Programming Languages Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996. ACM Press.

[TO98] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, June 1997.

[TSS00] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *International Conference on Functional Programming*, pages 82–93. ACM Press, September 2000.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

[VC03] Joseph C. Vanderwaart and Karl Crary. A typed interface for garbage collection. In *Types in Language Design and Implementation*, New Orleans, LA, January 2003.

[WA99] Daniel C. Wang and Andrew W. Appel. Safe garbage collection = regions + intensional type analysis. Technical Report TR-609-99, Princeton University, 1999.

[WA00] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors (extended version). Technical Report TR-624-00, Princeton University, 2000.

[WA01] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *Symposium on Principles of Programming Languages*, January 2001.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Symposium on Principles of Programming Languages*, Austin, TX, January 1989.

[Wer94] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, September 1992.

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures. In *International Workshop on Types in Compilation*, August 2000.

Appendix A

Soundness proofs

A.1 Soundness of principled scavenging

Throughout this section, I assume unique variable names. The environments are sets with no duplicate occurrences and no ordering. It is easy to show by induction over judgments that extending environments with additional bindings is safe. I will assume this in the rest of the section.

A.1.1 Soundness of λ_{GC}

The code region cd is always implicitly part of the environment. We treat it as a constant region. Even when the environment is restricted to a particular set, say $\Psi|_{\Theta}$, the code region is included in the restricted set. Therefore $\Psi|_{\nu_1,\ldots\nu_k}$ is equivalent to $\{\mathsf{cd}:\Upsilon_{\mathsf{cd}}, \nu_1:\Upsilon_{\nu_1}, \ldots \nu_k:\Upsilon_{\nu_k}\}$. And $\Psi|_{\mathsf{cd}}$ is equivalent to $\{\mathsf{cd}:\Upsilon_{\mathsf{cd}}\}$.

Lemma A.1.1 *If* $\Theta', r; \Delta; \Phi \vdash \sigma$*, then* $\Theta[^{\nu}/_{r}]; \Delta; \Phi[^{\nu}/_{r}] \vdash \sigma[^{\nu}/_{r}]$ *where* $\Theta', r = \Theta$*.*

Proof The proof is a straightforward induction over the structure of σ. □

Lemma A.1.2 $(\Phi[^{\nu}/_{r}])|_{\Theta,\nu} = (\Phi|_{\Theta,r})[^{\nu}/_{r}]$ *and* $(\Gamma[^{\nu}/_{r}])|_{\Theta,\nu} = (\Gamma|_{\Theta,r})[^{\nu}/_{r}]$

Proof The lemma is proved by considering the structure of Φ and Γ respectively, and comparing the results in the two cases. □

Lemma A.1.3 *If* $\Psi; \Theta', r; \Delta; \Phi; \Gamma \vdash op \,:\, \sigma$*, then*
$\Psi; \Theta[{}^{\nu}/_{r}]; \Delta; \Phi[{}^{\nu}/_{r}]; \Gamma[{}^{\nu}/_{r}] \vdash op[{}^{\nu}/_{r}] \,:\, \sigma[{}^{\nu}/_{r}]$ *where* $\Theta', r = \Theta$.

Proof The proof is by induction over the structure of *op*. Most of the cases follow directly by induction. We will show only the case for type packages.
case $\langle \alpha \!:\! \Theta_1 \!=\! \sigma_1, v \!:\! \sigma_2 \rangle$: We know that

$$\Psi; \Theta', r; \Delta; \Phi; \Gamma \vdash \langle \alpha \!:\! \Theta_1 \!=\! \sigma_1, v \!:\! \sigma_2 \rangle \,:\, \exists \alpha \!:\! \Theta_1 . \sigma_2$$

This implies that $\Theta_1; \Delta; \Phi|_{\Theta_1} \vdash \sigma_1$ and

$$\Psi; \Theta', r; \Delta; \Phi; \Gamma \vdash v \,:\, \sigma_2[{}^{\sigma_1}/_{\alpha}]$$

Suppose $r \notin \Theta_1$. Then r does not occur free in σ_1. Then applying the inductive hypothesis to the derivation for v, we get that

$$\Psi; \Theta[{}^{\nu}/_{r}]; \Delta; \Phi[{}^{\nu}/_{r}]; \Gamma[{}^{\nu}/_{r}] \vdash v[{}^{\nu}/_{r}] \,:\, (\sigma_2[{}^{\nu}/_{r}])[{}^{\sigma_1}/_{\alpha}]$$

Suppose $\Phi_1 = \Phi|_{\Theta_1}$. Then we have that $\Phi[{}^{\nu}/_{r}]|_{\Theta_1} = \Phi_1, \Phi_2$ and $Dom(\Phi_1) \cap Dom(\Phi_2) = \emptyset$. Therefore, we have that

$$\Theta_1; \Delta; \Phi|_{\Theta_1}, \Phi_2 \vdash \sigma_1$$

This implies that $\Theta_1; \Delta; \Phi[{}^{\nu}/_{r}]|_{\Theta_1} \vdash \sigma_1$, which leads to the required result.
Consider now that $r \in \Theta_1$. Suppose $\Theta_1 = \Theta_2, r$. Then

$$\Theta_2, r; \Delta; \Phi|_{\Theta_2, r} \vdash \sigma_1$$

Applying lemmas A.1.1 and A.1.2 we get that

$$\Theta_1[{}^{\nu}/_{r}]; \Delta; (\Phi[{}^{\nu}/_{r}])|_{\Theta_2, \nu} \vdash \sigma_1[{}^{\nu}/_{r}]$$

But $\Theta_2, \nu = \Theta_1[^{\nu}/_{r}]$. The second subderivation now becomes

$$\Psi; \Theta', r; \Delta; \Phi; \Gamma \vdash v : \sigma_2[^{\sigma_1}/_{\alpha}]$$

By applying the inductive hypothesis we get that

$$\Psi; \Theta[^{\nu}/_{r}]; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash v[^{\nu}/_{r}] : (\sigma_2[^{\nu}/_{r}])[^{\sigma_1[^{\nu}/_{r}]}/_{\alpha}]$$

This leads to the required result. $\square$

Lemma A.1.4 *If* $\Psi; \Theta', r; \Delta; \Phi; \Gamma \vdash e$, *then* $\Psi; \Theta[^{\nu}/_{r}]; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash e[^{\nu}/_{r}]$ *where* $\Theta', r = \Theta$

Proof The proof is by induction over the derivation of e. Most of the cases follow directly from the inductive hypothesis. We will consider only one case here.

case $\mathsf{only}\ \Theta_1\ \mathsf{in}\ e$: We get that

$$\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash \mathsf{only}\ \Theta_1\ \mathsf{in}\ e$$

This implies that

$$\Psi|_{\Theta_1}; \Theta_1, \mathsf{cd}; \Delta; \Phi|_{\Theta_1}; \Gamma|_{\Theta_1} \vdash e$$

and $\Theta_1 \subset \Theta, r$. Suppose $r \notin \Theta_1$. Then r does not occur free in e. Also $\Theta_1[^{\nu}/_{r}] = \Theta_1$. Suppose $\Gamma|_{\Theta_1} = \Gamma_1$. Then we have that $\Gamma[^{\nu}/_{r}]|_{\Theta_1} = \Gamma_1, \Gamma_2$ and $Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset$. Since we can extend environments, we get that $\Psi|_{\Theta_1}; \Theta_1, \mathsf{cd}; \Delta; \Phi_1, \Phi_2; \Gamma_1, \Gamma_2 \vdash e$, where Φ_1 and Φ_2 are constructed similar to Γ_1 and Γ_2. This implies that

$$\Psi|_{\Theta_1}; \Theta_1, \mathsf{cd}; \Delta; \Phi_1, \Phi_2; \Gamma_1, \Gamma_2 \vdash e$$

Also $\Theta_1 \subset (\Theta, r)[^{\nu}/_{r}]$. This leads to the required result.

Suppose now that $r \in \Theta_1$. Suppose that $\Theta_1 = \Theta_2, r$. Then $\Theta_1[^{\nu}/_{r}] = \Theta_2, \nu$. Then we have that

$$\Psi|_{\Theta_2,r}; \Theta_2, r, \mathsf{cd}; \Delta; \Phi|_{\Theta_2,r}; \Gamma|_{\Theta_2,r} \vdash e$$

Applying the inductive hypothesis we get that

$$\Psi|_{\Theta_2,r}; \Theta_1[^{\nu}/_{r}], \mathsf{cd}; \Delta; \Phi|_{\Theta_2,r}[^{\nu}/_{r}]; \Gamma|_{\Theta_2,r}[^{\nu}/_{r}] \vdash e[^{\nu}/_{r}]$$

Applying lemma A.1.2 we get that

$$\Psi|_{\Theta_2,r}; \Theta_1[^{\nu}/_{r}], \mathsf{cd}; \Delta; \Phi[^{\nu}/_{r}]|_{\Theta_2,\nu}; \Gamma[^{\nu}/_{r}]|_{\Theta_2,\nu} \vdash e[^{\nu}/_{r}]$$

But we have that $\Psi|_{\Theta_2,r} = \Psi|_{\Theta_2}$. Moreover, $\Psi|_{\Theta_2,\nu} = \Psi|_{\Theta_2}, \Psi'$. Therefore, we get that

$$\Psi|_{\Theta_2,\nu}; \Theta_1[^{\nu}/_{r}], \mathsf{cd}; \Delta; \Phi[^{\nu}/_{r}]|_{\Theta_2,\nu}; \Gamma[^{\nu}/_{r}]|_{\Theta_2,\nu} \vdash e[^{\nu}/_{r}]$$

We also have that $\Theta_1[^{\nu}/_{r}] \subset \Theta[^{\nu}/_{r}]$. This leads to the required result. □

Lemma A.1.5 *If* $\Delta, t{:}\kappa' \vdash \tau : \kappa$ *and* $\Delta \vdash \tau' : \kappa'$, *then* $\Delta \vdash \tau[^{\tau'}/_{t}] : \kappa$

Proof The proof is a straightforward induction over the structure of τ. □

Lemma A.1.6 *If* $\Theta; \Delta, t{:}\kappa; \Phi \vdash \sigma$ *and* $\Delta \vdash \tau : \kappa$, *then* $\Theta; \Delta; \Phi \vdash \sigma[^{\tau}/_{t}]$

Proof The proof is again a straighforward induction over the structure of σ. □

Lemma A.1.7 *If* $\Psi; \Theta; \Delta, t{:}\kappa; \Phi; \Gamma \vdash op : \sigma$ *and* $\Delta \vdash \tau : \kappa$ *then*
$\Psi; \Theta; \Delta; \Phi; \Gamma[^{\tau}/_{t}] \vdash op[^{\tau}/_{t}] : \sigma[^{\tau}/_{t}]$

Proof The proof is a straightforward induction over the structure of op. The only unusual case is when $op = \nu.\ell$. In this case, $\Psi(\nu.\ell) = \sigma$ and $Dom(\Psi); \cdot; \cdot \vdash \sigma$. Therefore, the variable t does not occur free in σ $\mathtt{at}$ ν. □

Lemma A.1.8 *If* $\Psi; \Theta; \Delta, t{:}\kappa; \Phi; \Gamma \vdash e$ *and* $\cdot \vdash \tau' : \kappa$ *then* $\Psi; \Theta; \Delta; \Phi; \Gamma[^{\tau'}/_{t}] \vdash e[^{\tau'}/_{t}]$

Proof The proof is a straightforward induction over the structure of e. The only interesting case is for a tcase when the substituted variable is being analyzed.

case $\mathsf{tcase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists})$: Suppose we substitute the type τ' for the variable t. Then τ' can only be one of Int, $\tau'' \rightarrow 0$, $\tau_1' \times \tau_2'$, or $\exists t.\tau''$. For a Int, we need to prove that

$$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\mathsf{Int}}/_{t}] \vdash (\mathsf{tcase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists}))[{}^{\mathsf{Int}}/_{t}]$$

This implies that we need to prove that

$$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\mathsf{Int}}/_{t}] \vdash e_i[{}^{\mathsf{Int}}/_{t}]$$

By definition, we know that

$$\Psi; \Theta; \Delta, t\!:\!\Omega; \Phi; \Gamma[{}^{\mathsf{Int}}/_{t}] \vdash e_i[{}^{\mathsf{Int}}/_{t}]$$

Since t is being substituted away, this leads to the required result.

For a code type we need to prove that

$$\begin{aligned}&\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\tau' \rightarrow 0}/_{t}] \vdash \\ &\quad (\mathsf{tcase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists}))[{}^{\tau' \rightarrow 0}/_{t}]\end{aligned}$$

This implies that we need to prove that

$$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\tau' \rightarrow 0}/_{t}] \vdash e_{\rightarrow}[{}^{\tau' \rightarrow 0}/_{t}]$$

By definition, we get that $\Psi; \Theta; \Delta, t\!:\!\Omega; \Phi; \Gamma \vdash e_{\rightarrow}$. Substituting for t and applying the inductive hypothesis leads to the result.

For the pair type we need to prove that

$$\begin{aligned}&\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{(\tau_1' \times^{\rho} \tau_2')}/_{t}] \vdash \\ &\quad (\mathsf{tcase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists}))[{}^{(\tau_1' \times^{\rho} \tau_2')}/_{t}]\end{aligned}$$

This implies that we need to prove that

$$\Psi;\Theta;\Delta;\Phi;\Gamma[{}^{(\tau_1' \times^\rho \tau_2')}/_t] \vdash e_\times[{}^{(\tau_1' \times^\rho \tau_2'), \tau_1', \tau_2'}/_{t, t_1, t_2}]$$

By definition, we know that

$$\Psi;\Theta;\Delta, t:\Omega, t_1:\Omega, t_2:\Omega;\Phi;\Gamma[{}^{t_1 \times^r t_2}/_t] \vdash e_\times[{}^{t_1 \times^r t_2}/_t]$$

Note that the variables t_1 and t_2 do not occur free separately in Γ. Substituting τ_1' for t_1, τ_2' for t_2, and $\tau_1' \times^r \tau_2'$ for $t_1 \times^r t_2$ leads to the required result.
For the existential type we need to prove that

$$\begin{aligned}&\Psi;\Theta;\Delta;\Phi;\Gamma[{}^{\exists t_1.\tau'}/_t] \vdash \\ &\quad (\mathsf{tcase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_\to; t_1 t_2.e_\times; t_e.e_\exists))[{}^{\exists t_1.\tau'}/_t]\end{aligned}$$

This implies that we need to prove that

$$\Psi;\Theta;\Delta;\Phi;\Gamma[{}^{\exists t_1.\tau'}/_t] \vdash e_\exists[{}^{\exists t_1.\tau', \lambda t_1.\tau'}/_{t, t_e}]$$

By definition we know that

$$\Psi;\Theta;\Delta, t:\Omega, t_e:\Omega \to \Omega;\Phi;\Gamma[{}^{\exists t_1.t_e t_1}/_t] \vdash e_\exists[{}^{\exists t_1.t_e t_1}/_t]$$

Substituting $(\lambda t_1.\tau')$ for t_e and applying the inductive hypothesis we get that

$$\Psi;\Theta;\Delta, t:\Omega;\Phi;\Gamma[{}^{\exists t_1.\tau'}/_t] \vdash e_\exists[{}^{\exists t_1.\tau', \lambda t_1.\tau'}/_{t, t_e}]$$

Since t is being substituted away, we can remove it from the type environment. This leads to the required result. $\square$

Lemma A.1.9 *If* $\Theta;\Delta;\Phi,\alpha:\Theta' \vdash \sigma$ *and* $\Theta';\Delta;\Phi \vdash \sigma'$, *then* $\Theta;\Delta;\Phi \vdash \sigma[{}^{\sigma'}/_\alpha]$

Proof The proof is a straighforward induction over the structure of σ. In the case of c'ode types, we use the fact that the argument types $\vec{\sigma}$ are fully closed. □

Lemma A.1.10 *If* $\Psi; \Theta; \Delta; \Phi, \alpha : \Theta'; \Gamma \vdash op : \sigma$ *and* $\Theta'; \Delta; \Phi \vdash \sigma'$ *then* $\Psi; \Theta; \Delta; \Phi; \Gamma[^{\sigma'}/_{\alpha}] \vdash op[^{\sigma'}/_{\alpha}] : \sigma[^{\sigma'}/_{\alpha}]$

Proof The proof is again by induction over the typing derivation for op. We will consider only the case for packages.

case $\langle \beta : \Theta' = \sigma_1, v : \sigma_2 \rangle$: By definition,

$$\Psi; \Theta; \Delta; \Phi, \alpha : \Theta_1; \Gamma \vdash \langle \beta : \Theta' = \sigma_1, v : \sigma_2 \rangle : \exists \beta : \Theta'.\sigma_2$$

There are two possible cases. If $\Theta_1 \subset \Theta'$, then we get that $\Theta'; \Delta; \Phi|_{\Theta'}, \alpha : \Theta_1 \vdash \sigma_1$ and $\Psi; \Theta; \Delta; \Phi, \alpha : \Theta_1; \Gamma \vdash v : \sigma_2[^{\sigma_1}/_{\beta}]$ By lemma A.1.9, $\Theta'; \Delta; \Phi|_{\Theta'} \vdash \sigma_1[^{\sigma'}/_{\alpha}]$ Applying the inductive hypothesis on the typing rule for v leads to the result.

In the other case, we get that $\Theta'; \Delta; \Phi|_{\Theta'} \vdash \sigma_1$ and $\Psi; \Theta; \Delta; \Phi, \alpha : \Theta_1; \Gamma \vdash v : \sigma_2[^{\sigma_1}/_{\beta}]$ This implies that α does not occur free in σ_1. Therefore, we need to prove that

$$\begin{aligned}&\Psi; \Theta; \Delta; \Phi; \Gamma[^{\sigma'}/_{\alpha}] \vdash \\ &\quad \langle \beta : \Theta' = \sigma_1, v[^{\sigma'}/_{\alpha}] : \sigma_2[^{\sigma'}/_{\alpha}] \rangle : \exists \beta : \Theta'.\sigma_2[^{\sigma'}/_{\alpha}]\end{aligned}$$

This follows from applying the inductive hypothesis to the judgment for v. □

Lemma A.1.11 *If* $\Theta; \Delta; \Phi \vdash \sigma[^{\sigma'}/_{\alpha}]$ *and* α *occurs free in* σ*, then* $\Theta; \Delta; \Phi \vdash \sigma'$

Proof The proof is by induction over the structure of σ. □

Lemma A.1.12 *If* $\Psi; \Theta; \Delta; \Phi, \alpha : \Theta'; \Gamma \vdash e$ *and* $\Theta'; \Delta; \Phi \vdash \sigma'$ *then* $\Psi; \Theta; \Delta; \Phi; \Gamma[^{\sigma'}/_{\alpha}] \vdash e[^{\sigma'}/_{\alpha}]$

Proof The proof is again by induction over the derivation. The only non-trivial case is the only construct.

case $\mathsf{only}\ \Theta_1\ \mathsf{in}\ e$: By definition, $\Psi;\Theta;\Delta;\Phi,\alpha\!:\!\Theta';\Gamma \vdash \mathsf{only}\ \Theta_1\ \mathsf{in}\ e$ and $\Theta';\Delta;\Phi \vdash \sigma'$. Suppose $\Theta' \subset \Theta_1$. Then we get that

$$\Psi|_{\Theta_1};\Theta_1,\mathsf{cd};\Delta;\Phi|_{\Theta_1},\alpha\!:\!\Theta';\Gamma|_{\Theta_1} \vdash e$$

Applying the inductive hypothesis we get that

$$\Psi|_{\Theta_1};\Theta_1,\mathsf{cd};\Delta;\Phi|_{\Theta_1};\Gamma|_{\Theta_1}[{}^{\sigma'}/_{\alpha}] \vdash e[{}^{\sigma'}/_{\alpha}]$$

But we also have that $\Gamma|_{\Theta_1}[{}^{\sigma'}/_{\alpha}] = \Gamma[{}^{\sigma'}/_{\alpha}]|_{\Theta_1}$. From here we can conclude that

$$\Psi;\Theta;\Delta;\Phi;\Gamma[{}^{\sigma'}/_{\alpha}] \vdash \mathsf{only}\ \Theta_1\ \mathsf{in}\ e[{}^{\sigma'}/_{\alpha}]$$

If $\Theta_1 \subset \Theta'$, then we get that

$$\Psi|_{\Theta_1};\Theta_1,\mathsf{cd};\Delta;\Phi|_{\Theta_1};\Gamma|_{\Theta_1} \vdash e$$

This implies that α does not occur free in e. We also have that $\Theta';\Delta;\Phi \vdash \sigma'$. Using lemma A.1.11, we can show that $\Gamma[{}^{\sigma'}/_{\alpha}]|_{\Theta_1} = \Gamma|_{\Theta_1}$. Therefore, we get that

$$\Psi|_{\Theta_1};\Theta_1,\mathsf{cd};\Delta;\Phi|_{\Theta_1};\Gamma[{}^{\sigma'}/_{\alpha}]|_{\Theta_1} \vdash e$$

This implies that

$$\Psi;\Theta;\Delta;\Phi;\Gamma[{}^{\sigma'}/_{\alpha}] \vdash \mathsf{only}\ \Theta_1\ \mathsf{in}\ e$$

□

Lemma A.1.13 *If* $\Psi;\Theta;\Delta;\Phi;\Gamma,x\!:\!\sigma' \vdash op\ :\ \sigma$ *and* $\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v'\ :\ \sigma'$ *then* $\Psi;\Theta;\Delta;\Phi;\Gamma \vdash op[{}^{v'}/_{x}]\ :\ \sigma$

Proof The proof is a straightforward induction over the typing derivation for op. □

Lemma A.1.14 *If* $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma$ *and* $\Theta_1; \Delta; \Phi|_{\Theta_1} \vdash \sigma$ *and* $\Theta_1 \subset \Theta$*, then* $\Psi|_{\Theta_1}; \Theta_1; \Delta; \Phi|_{\Theta_1}; \Gamma|_{\Theta_1} \vdash v : \sigma$

Proof The proof is by induction over the derivation for v. Most of the cases follow directly from the inductive hypothesis. We will consider only one case here.

case $\nu.\ell$: We have that $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash \nu.\ell : \sigma \text{ at } \nu$. This implies that $\Psi(\nu.\ell) = \sigma$ and $Dom(\Psi); \cdot; \cdot \vdash \sigma \text{ at } \nu$. However, by assumption we also know that $\Theta_1; \Delta; \Phi|_{\Theta_1} \vdash \sigma \text{ at } \nu$. This implies that $\nu \in \Theta_1$. This implies that $\Psi|_{\Theta_1}(\nu.\ell) = \sigma$. Moreover, we also get that $\Theta_1; \cdot; \cdot \vdash \sigma \text{ at } \nu$. Therefore, we get that $Dom(\Psi|_{\Theta_1}); \cdot; \cdot \vdash \sigma \text{ at } \nu$. From here we get that $\Psi|_{\Theta_1}; \Theta_1; \Delta; \Phi|_{\Theta_1}; \Gamma|_{\Theta_1} \vdash \nu.\ell : \sigma \text{ at } \nu$. □

Lemma A.1.15 *If* $\Psi; \Theta; \Delta; \Phi; \Gamma, x{:}\sigma \vdash e$ *and* $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma$ *then* $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash e[{}^{v}\!/_{x}]$

Proof The proof is again a straightforward induction over the structure of e. We will only show the proof for a couple of cases, the rest of them follow similarly.

case only Θ in e: We have that $\Psi; \Theta; \Delta; \Phi; \Gamma, x{:}\sigma \vdash \text{only } \Theta_1 \text{ in } e$. This implies that $\Psi|_{\Theta_1}; \Theta_1; \Delta; \Phi|_{\Theta_1}; (\Gamma, x{:}\sigma)|_{\Theta_1} \vdash e$. If we have that $\Theta_1; \Delta; \Phi|_{\Theta_1} \vdash \sigma$, then we get that $\Psi|_{\Theta_1}; \Theta_1; \Delta; \Phi|_{\Theta_1}; \Gamma|_{\Theta_1}, x{:}\sigma \vdash e$. By lemma A.1.14 we get that $\Psi|_{\Theta_1}; \Theta_1; \Delta; \Phi|_{\Theta_1}; \Gamma|_{\Theta_1} \vdash v : \sigma$. Applying the inductive hypothesis gives us that $\Psi|_{\Theta_1}; \Theta_1; \Delta; \Phi|_{\Theta_1}; \Gamma|_{\Theta_1} \vdash e[{}^{v}\!/_{x}]$.

In the other case, we get that $\Psi|_{\Theta_1}; \Theta_1; \Delta; \Phi|_{\Theta_1}; \Gamma|_{\Theta_1} \vdash e$. This implies that x does not occur free in e. The required result follows from here.

case $\mathsf{tcase}\ t\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; t_1 t_2.e_{\times}; t_e.e_{\exists})$: By assumption, we get that

$\Delta \vdash t : \Omega$

$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\mathsf{Int}}/_{t}], x : \sigma[{}^{\mathsf{Int}}/_{t}] \vdash e_i[{}^{\mathsf{Int}}/_{t}]$

$\Psi; \Theta; \Delta; \Phi; \Gamma, x : \sigma \vdash e_{\rightarrow}$

$\Psi; \Theta; \Delta, t_1 : \Omega, t_2 : \Omega; \Phi; \Gamma[{}^{t_1 \times^r t_2}/_{t}], x : \sigma[{}^{t_1 \times^r t_2}/_{t}] \vdash$
$e_{\times}[{}^{t_1 \times^r t_2}/_{t}]$

$\Psi; \Theta; \Delta, t_e : \Omega \rightarrow \Omega; \Phi; \Gamma[{}^{\exists t_1.t_e t_1}/_{t}], x : \sigma[{}^{\exists t_1.t_e t_1}/_{t}] \vdash$
$e_{\exists}[{}^{\exists t_1.t_e t_1}/_{t}]$

By lemma A.1.7, we know that if $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma$, then $\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\tau}/_{t}] \vdash v : \sigma[{}^{\tau}/_{t}]$. Now substitute $v[{}^{\mathsf{Int}}/_{t}]$ in the e_i branch, substitute v in the $e_{\rightarrow}$ branch, substitute $v[{}^{t_1 \times^r t_2}/_{t}]$ in the $e_{\times}$ branch, and substitute $v[{}^{\exists t_1.t_e t_1}/_{t}]$ in the $e_{\exists}$ branch. The required result follows from the inductive hypothesis on each branch. □

Proposition A.1.16 (Type Preservation) *If* $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.

Proof The proof is by induction over the evaluation relation. We will consider only the cases that do not follow directly from the inductive hypothesis and the substitution lemmas.

case $\nu.\ell[\vec{\tau}][\vec{\nu}](\vec{v})$: The lemma follows from the fact that tag reduction is strongly normalizing and confluent, and that tag reduction preserves kind.

case $\nu.\ell[\vec{\tau'}][\vec{\nu}](\vec{v})$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \nu.\ell[\vec{\tau'}][\vec{\nu}](\vec{v})$$

Since $M(\nu.\ell) = (\lambda[\vec{t : \kappa}][\vec{r}](\vec{x} : \vec{\sigma}).e)$, we have that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \nu.\ell : \forall[\vec{t : \kappa}][\vec{r}](\vec{\sigma}) \rightarrow 0 \text{ at } \nu$$

This implies that

$$\Psi|_{\mathsf{cd}}; \mathsf{cd}, \vec{r}; \overrightarrow{t : \kappa}; \cdot; \overrightarrow{x : \sigma} \vdash e$$

By the typing rule, we get that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_i : \sigma_i[{}^{\vec{\nu}, \vec{\tau'}}/_{\vec{r}, \vec{t}}]$$

and $\cdot \vdash \tau'_i : \kappa_i$. From lemma A.1.4 we get that

$$\Psi|_{\mathsf{cd}}; \mathsf{cd}, \vec{\nu}; \Delta; \cdot; \overrightarrow{x : \sigma[{}^{\vec{\nu}}/_{\vec{r}}]} \vdash e[{}^{\vec{\nu}}/_{\vec{r}}]$$

From lemma A.1.8 we get that

$$\Psi|_{\mathsf{cd}}; \mathsf{cd}, \vec{\nu}; \cdot; \cdot; \overrightarrow{x : \sigma[{}^{\vec{\nu}, \vec{\tau'}}/_{\vec{r}, \vec{t}}]} \vdash e[{}^{\vec{\nu}, \vec{\tau'}}/_{\vec{r}, \vec{t}}]$$

Since $\Psi|_{\mathsf{cd}} \subset \Psi$ and $\mathsf{cd}, \vec{\nu} \subset Dom(\Psi)$, we can extend the environment for deriving e. Applying lemma A.1.15 we get that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash e[\vec{\nu}, \vec{\tau'}, \vec{v}/\vec{r}, \vec{t}, \vec{x}]$$

which leads to the result.

case $(v[\![\vec{\tau}]\!])[\vec{\tau}][\vec{\nu}](\vec{v})$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash (v[\![\vec{\tau}]\!])[\vec{\tau}][\vec{\nu}](\vec{v})$$

From the typing rules $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash (v[\![\vec{\tau}]\!]) : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\nu} 0$ for some ν and $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_i : \sigma_i[{}^{\vec{\nu}}/_{\vec{r}}]$. Again from the typing rules we get that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \forall[\vec{t : \kappa}][\vec{r}](\vec{\sigma'}) \to 0 \text{ at } \nu$ where $\sigma'_i[{}^{\vec{\tau}}/_{\vec{t}}] = \sigma_i$ and $\cdot \vdash \tau_i : \kappa_i$. We need to prove that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v[\vec{\tau}][\vec{\nu}](\vec{v})$$

This is true if $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_i : \sigma'_i[{}^{\vec{\nu}, \vec{\tau}}/_{\vec{r}, \vec{t}}]$. But we already know that this holds.

case $\mathsf{let}\ x = \mathsf{put}[\nu]v\ \mathsf{in}\ e$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{let}\ x = \mathsf{put}[\nu]v\ \mathsf{in}\ e$$

From the typing rules,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{put}[\nu]v\ :\ \sigma\ \mathsf{at}\ \nu$$

for some type σ, and $\nu \in Dom(\Psi)$. This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v\ :\ \sigma$$

Again, from the typing rules,

$$\Psi, \nu.\ell : \sigma; Dom(\Psi); \cdot; \cdot; \cdot \vdash \nu.\ell\ :\ \sigma\ \mathsf{at}\ \nu$$

The required result now follows from lemma A.1.15.

case $\mathsf{let}\ x = \mathsf{get}\ \nu.\ell\ \mathsf{in}\ e$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{let}\ x = \mathsf{get}\ \nu.\ell\ \mathsf{in}\ e$$

From the typing rules we get that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \nu.\ell\ :\ \sigma\ \mathsf{at}\ \nu$$

for some type σ Again from the typing rules, we get that $\Psi(\nu.\ell) = \sigma$. This implies that if $M(\nu.\ell) = v$, then

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v\ :\ \sigma$$

The required result follows from lemma A.1.15.

case $\mathsf{open}\ \langle \alpha : \Theta = \sigma_1, v : \sigma_2 \rangle\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e$: The two open constructs are proved similarly. We will show the proof for only one of them. By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{open}\ \langle \alpha : \Theta = \sigma_1, v : \sigma_2 \rangle\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \alpha : \Theta; x : \sigma_2 \vdash e$$

The required result follows from lemmas A.1.15 and A.1.12.

case $\mathsf{let\ region}\ r\ \mathsf{in}\ e$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{let\ region}\ r\ \mathsf{in}\ e$$

This implies that

$$\Psi; Dom(\Psi), r; \cdot; \cdot; \cdot \vdash e$$

By lemma A.1.4,

$$\Psi; Dom(\Psi), \nu; \cdot; \cdot; \cdot \vdash e[{}^{\nu}/_{r}]$$

Since ν is a newly introduced region, we can extend Ψ with it. This implies that

$$\Psi, \nu \mapsto \{\}; Dom(\Psi), \nu; \cdot; \cdot; \cdot \vdash e[{}^{\nu}/_{r}]$$

This is the required result.

case $\mathsf{only}\ \Theta\ \mathsf{in}\ e$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{only}\ \Theta\ \mathsf{in}\ e$$

This implies that

$$\Psi|_{\Theta}; \mathsf{cd}, \Theta; \cdot; \cdot; \cdot \vdash e$$

But $\mathsf{cd}, \Theta = Dom(\Psi|_\Theta)$. This implies that

$$\Psi|_\Theta; Dom(\Psi|_\Theta); \cdot; \cdot; \cdot \vdash e$$

which is the required result.

For all of the tcase, the required result follows directly from the typing rules since the value environment is empty.

Lemma A.1.17 (Canonical forms)

1. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \ : \ \mathit{int}$ *then* $v = n$.
2. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \ : \ \sigma \ \mathsf{at}\ \nu$ *then* $v = \nu.\ell$.
3. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \ : \ \sigma_1 \times \sigma_2$ *then* $v = \langle v_1, v_2 \rangle$.
4. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \ : \ \exists t : \kappa.\sigma$ *then* $v = \langle t = \tau, v' : \sigma \rangle$.
5. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \ : \ \exists \alpha : \Theta'.\sigma$ *then* $v = \langle \alpha : \Theta' = \sigma_1, v' : \sigma_2 \rangle$.
6. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \ : \ \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0$ *then* $v = v'[\![\vec{\tau}]\!]$.
7. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \ : \ \forall[\vec{t : \kappa}][\vec{r}](\vec{\sigma}) \rightarrow 0$ *then* $v = \lambda[\vec{t : \kappa}][\vec{r}](\vec{x : \sigma}).e$.

Proof The proof follows from the inspection of the typing rules for values. □

Proposition A.1.18 (Progress) *If* $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* (M', e') *such that* $(M, e) \Longrightarrow (M', e')$.

Proof The proof is again by induction over the structure of e. By definition, $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash e$. The proof for the individual cases start from this point.

case $v[\vec{\tau}][\vec{\nu}](\vec{v})$: From the typing rules, either

$$v : \forall[\vec{t : \kappa}][\vec{r}](\vec{\sigma}) \rightarrow 0 \ \mathsf{at}\ \nu \quad \mathit{or} \quad v : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\nu} 0$$

In the first case by lemma A.1.17, $v = \nu.\ell$. From the typing rules $M(\nu.\ell) = \lambda[\vec{t : \kappa}][\vec{r}](\vec{x : \sigma}).e$ This implies that we have a reduction.

In the second case, by lemma A.1.17 $v = v'[\![\tau]\!]$. In this case also we have a reduction to $v'[\vec{\tau}][\vec{\nu}](\vec{v})$.

case let $x = op$ in e: If $op = v$, then we have a reduction. If $op = \pi_i v$, then from the typing rules, $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \sigma_1 \times \sigma_2$. The required result follows from lemma A.1.17. In the case of put$[\nu]v$, the result follows directly. The constraint $\nu \in \Theta$ ensures that $\nu \in Dom(\Psi)$. In the case for get v, by the typing rules we know that $v = \nu.\ell$ for some $\nu.\ell$. Again from the typing rule we know that $\Psi(\nu.\ell) = \sigma$. This implies that $M(\nu.\ell) = v'$ for some value v'.

For the other cases of e, the proposition follows directly from the operational semantics. □

A.1.2 Soundness of $\lambda_{\mathsf{GCforw}}$

As before, We use the usual exchange and weakening properties of environments without proving them. The code region cd is always implicitly part of the environment. Even when the environment is restricted to a particular set, say $\Psi|_\Theta$, the code region is included in the restricted set.

We will show the proofs only for the extensions. The only non-trivial extension is the widen construct. The proofs of the propositions for the other constructs follow in a straightforward way by induction over the derivations.

Because of the widen operator, the notion of a well formed machine state has to be slightly adjusted to allow non-well-formed memory elements as long as they can be shown not to influence the execution:

Definition A.1.19 *The machine state* (M, e) *is well formed iff*

$$\frac{\overline{M} \subset M \qquad \vdash \overline{M} : \Psi \qquad \Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash e}{\vdash (M, e)}$$

Note that to show well-formedness we now need to find not only a witness Ψ but also a proper $\overline{M}$ subset.

The reason is that widen casts the whole live heap from one type to the other but cannot cast any arbitrary M because it might include existential packages of the form $\langle \alpha : \Theta = \sigma_1, v : \sigma_2 \rangle$ where casting α would require an update of M, which goes against the idea that widen is a nop.

This problem appears when we try to prove type preservation, which is the main obstacle to show soundness of the widen operator, whose proof uses two lemmas for the following two steps:

1. First, we construct a $\overline{M}$ which only contains elements that we can cast (i.e. of type $\mathsf{M}_\rho(\tau)$) and we show that the state is still well-formed. This is the case because when we reach a widen, all the live data if of such a type.

2. Then we show that when everything has a type of the form $\mathsf{M}_\rho(\tau)$, we can cast every single type consistently to its $\mathsf{C}_{\rho,\rho'}(\tau)$ equivalent and the result is still properly typed. This relies on the subsumption rule on sum types and the fact that C types only differs from M types by adding branches to sum types.

Lemma A.1.20 (Region Substitution)

1. *If* $\Psi;\Theta,r;\Delta;\Phi;\Gamma \vdash op\ :\ \sigma$
   then $\Psi;\Theta,\nu;\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}] \vdash op[{}^{\nu}/_{r}]\ :\ \sigma[{}^{\nu}/_{r}]$.

2. *If* $\Psi;\Theta,r;\Delta;\Phi;\Gamma \vdash e$
   then $\Psi;\Theta,\nu;\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}] \vdash e[{}^{\nu}/_{r}]$.

Proof Straightforward proof via induction over the derivation of the typing judgment. Note that $\nu \in \Psi$ is not required here.

case $\mathsf{let}\ x = \mathsf{widen}[\rho'][\tau](v)\ \mathsf{in}\ e$: We have that $\Psi;\Theta,r;\Delta;\Phi;\Gamma \vdash \mathsf{let}\ x = \mathsf{widen}[\rho'][\tau](v)\ \mathsf{in}\ e$. This implies $\Psi;\Theta,r;\Delta;\Phi;\Gamma \vdash v\ :\ \mathsf{M}_\rho(\tau)$ and $\Psi|_{\mathsf{cd}};\mathsf{cd},\rho,\rho';\Delta;\Phi|_{\rho\rho'};x{:}\mathsf{C}_{\rho,\rho'}(\tau) \vdash e$. Applying lemma A.1.20.1 to the derivation for v we get that $\Psi;\Theta,\nu;\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}] \vdash v[{}^{\nu}/_{r}]\ :\ \mathsf{M}_{\rho[{}^{\nu}/_{r}]}(\tau)$ since $\mathsf{M}_\rho(\tau)[{}^{\nu}/_{r}] = \mathsf{M}_{\rho[{}^{\nu}/_{r}]}(\tau)$.
If either $\rho = r$ or $\rho' = r$ then the inductive hypothesis on the derivation of e and lemma A.1.2 and the fact that $\mathsf{C}_{\rho,\rho'}(\tau)[{}^{\nu}/_{r}] = \mathsf{C}_{(\rho,\rho')[{}^{\nu}/_{r}]}(\tau)$ leads to the required result.
In the other case, $\Phi[{}^{\nu}/_{r}]|_{\rho\rho'} = \Phi|_{\rho\rho'},\Phi'$ where $Dom(\Phi|_{\rho\rho'}) \cap Dom(\Phi')$ is empty. Moreover, r does not occur free in e. Since we can extend environments with new bindings we get that $\Psi|_{\mathsf{cd}};\mathsf{cd},\rho,\rho';\Delta;\Phi|_{\rho\rho'},\Phi';x{:}\mathsf{C}_{\rho,\rho'}(\tau) \vdash e$. This leads to the required result. □

Lemma A.1.21 (Type Substitution)

1. *If* $\Psi;\Theta;\Delta,t{:}\kappa;\Phi;\Gamma \vdash op\ :\ \sigma$ *and* $\Delta \vdash \tau\ :\ \kappa$
   then $\Psi;\Theta;\Delta;\Phi;\Gamma[{}^{\tau}/_{t}] \vdash op[{}^{\tau}/_{t}]\ :\ \sigma[{}^{\tau}/_{t}]$.

2. *If* $\Psi;\Theta;\Delta,t{:}\kappa;\Phi;\Gamma \vdash e$ *and* $\Delta \vdash \tau\ :\ \kappa$
   then $\Psi;\Theta;\Delta;\Phi;\Gamma[{}^{\tau}/_{t}] \vdash e[{}^{\tau}/_{t}]$.

3. *If* $\Psi; \Theta; \Delta; \Phi, \alpha : \Theta'; \Gamma \vdash op \ : \ \sigma$ *and* $\Theta'; \Delta; \Phi \vdash \sigma'$
   then $\Psi; \Theta; \Delta; \Phi; \Gamma[^{\sigma'}/_{\alpha}] \vdash op[^{\sigma'}/_{\alpha}] \ : \ \sigma[^{\sigma'}/_{\alpha}]$

4. *If* $\Psi; \Theta; \Delta; \Phi, \alpha : \Theta'; \Gamma \vdash e$ *and* $\Theta'; \Delta; \Phi \vdash \sigma'$
   then $\Psi; \Theta; \Delta; \Phi; \Gamma[^{\sigma'}/_{\alpha}] \vdash e[^{\sigma'}/_{\alpha}]$

Proof Same thing here, the induction hypothesis can always be applied directly. □

Lemma A.1.22 (Value Substitution)

1. *If* $\Psi; \Theta; \Delta; \Phi; \Gamma, x : \sigma \vdash op \ : \ \sigma'$
   and $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v \ : \ \sigma$
   then $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash op[^{v}/_{x}] \ : \ \sigma'$.

2. *If* $\Psi; \Theta; \Delta; \Phi; \Gamma, x : \sigma \vdash e$
   and $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v \ : \ \sigma$
   then $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash e[^{v}/_{x}]$.

Proof The proof is by induction over the derivation of the type judgment. Most of the cases follow directly from the induction hypothesis. Exceptions are:

widen We need to show that the substitution on e has no effect, which is easy since the variable we are substituting cannot occur freely in e.

□

Definition A.1.23 *Let* $\Psi|_{\nu,\mathsf{M}}$ *be the restriction of* Ψ *to pieces of code or elements of type* $\mathsf{M}_{\nu}(\tau)$*:*

$$\Psi|_{\nu',\mathsf{M}} = \{\nu.\ell : \sigma \mid \Psi(\nu.\ell) = \sigma \wedge \begin{pmatrix} \sigma = \forall[\vec{\tau}][\vec{r}](\overrightarrow{v : \sigma}) \rightarrow 0 \\ \vee \ \exists \tau.(\sigma \ \mathsf{at} \ \nu = \mathsf{M}_{\nu'}(\tau)) \end{pmatrix} \}$$

Note that since cd only contains functions, $\Psi|_{\nu,\mathsf{M}}$ will preserve all its contents.

Lemma A.1.24 *If* $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v \ : \ \mathsf{M}_{\nu}(\tau)$ *then* $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_{\nu}); \cdot; \cdot; \cdot \vdash v \ : \ \mathsf{M}_{\nu}(\tau)$

Proof By the definition of the M type, v is either of type int or of type σ at ν. The lemma follows trivially if v is of type int. Consider the other cases.

case $\mathsf{M}_\nu(\tau \to 0)$: In this case $v = \mathsf{cd}.\ell$. Therefore $\Psi(\mathsf{cd}.\ell) = \forall[][r](\mathsf{M}_r(\tau)) \to 0$. Therefore $M(\mathsf{cd}.\ell) = \lambda[][r](\overrightarrow{x\!:\!\sigma}).e$. The lemma now follows since the body e is typed only under the code region $\Psi|_{\mathsf{cd}}$ and the region environment cd, r.

case $\mathsf{M}_\nu(\tau_1 \times \tau_2)$: We again have that $v = \nu.\ell$. Therefore, $\Psi(\nu.\ell) = \mathsf{left}(\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$. This implies that $M(\nu.\ell) = \mathsf{inl}\ v'$. We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{inl}\ v' : \mathsf{left}(\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v' : (\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$$

This implies that $v' = \langle v_1', v_2' \rangle$. From the typing rules we get that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_1' : \mathsf{M}_\nu(\tau_1)$ and $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_2' : \mathsf{M}_\nu(\tau_2)$. Applying the inductive hypothesis to the derivations for v_1' and v_2', we get that $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v_1' : \mathsf{M}_\nu(\tau_1)$ and $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v_2' : \mathsf{M}_\nu(\tau_2)$. The required result follows from this.

case $\mathsf{M}_\nu(\exists t.\tau)$: We again have that $v = \nu.\ell$. Therefore, $\Psi(\nu.\ell) = \mathsf{left}(\exists t.\mathsf{M}_\nu(\tau))$. This implies that $M(\nu.\ell) = \mathsf{inl}\ v'$. We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{inl}\ v' : \mathsf{left}(\exists t.\mathsf{M}_\nu(\tau))$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v' : \exists t.\mathsf{M}_\nu(\tau)$$

This implies that $v' = \langle t{=}\tau', v''{:}\mathsf{M}_\nu(\tau)\rangle$. From here we get that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v'' : (\mathsf{M}_\nu(\tau))[{}^{\tau'}\!/_t]$. But we have that $(\mathsf{M}_\nu(\tau))[{}^{\tau'}\!/_t] = \mathsf{M}_\nu(\tau[{}^{\tau'}\!/_t])$. Therefore, applying the inductive hypothesis to the derivation for v'', we get that $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v'' : \mathsf{M}_\nu(\tau[{}^{\tau'}\!/_t])$. The required result follows from this. $\square$

Definition A.1.25 *Let $T_{\rho,\rho'}$ be the type operator that turns a type of the form $\mathsf{M}_\rho(\tau)$ into a type of the form $\mathsf{C}_{\rho,\rho'}(\tau)$ and keeps code pointer types unchanged:*

$$T_{\rho,\rho'}(\mathsf{M}_\rho(\tau)) = \mathsf{C}_{\rho,\rho'}(\tau)$$
$$T_{\rho,\rho'}(\forall[\vec{\tau}][\vec{r}](\overrightarrow{v\!:\!\sigma}) \to 0 \text{ at } \rho) = \forall[\vec{\tau}][\vec{r}](\overrightarrow{v\!:\!\sigma}) \to 0 \text{ at } \rho$$

We also use this operator on Ψ where it is defined as:

$$T_{\rho,\rho'}(\Psi) = \{\nu.\ell\!:\!\sigma \mid \Psi(\nu.\ell) = \sigma' \wedge T_{\rho,\rho'}(\sigma' \text{ at } \rho) = \sigma \text{ at } \rho\}$$

Note that the two parts of the definition of $T_{\rho,\rho'}(\sigma)$ overlap but are consistent since $\mathsf{M}_\rho(\tau)$ and $\mathsf{C}_{\rho,\rho'}(\tau)$ are identical in the case when τ is an arrow type. Also, since cd only contains functions, $T_{\rho,\rho'}(\Psi)$ does not change the type of cd. Finally since both M and C are iterators, the T type operator can be defined as an iterator.

Lemma A.1.26 *If $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v : \mathsf{M}_\nu(\tau)$, then $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash v : \mathsf{C}_{\nu\nu'}(\tau)$*

Proof When v is of type int, the lemma follows trivially.

case $\mathsf{M}_\nu(\tau \to 0)$: In this case $v = \mathsf{cd}.\ell$ and $M(\mathsf{cd}.\ell) = \lambda[][r](\overrightarrow{x\!:\!\sigma}).e$. This implies that $\Psi|_{\mathsf{cd}}; \mathsf{cd}, r; \cdot; \cdot; \overrightarrow{x\!:\!\sigma} \vdash e$. Since $\mathsf{M}_\nu(\tau' \to 0) = \mathsf{C}_{\nu\nu'}(\tau' \to 0)$ the type operator T is the identity for code types. Therefore, the cast will leave the type of the code region unchanged. Therefore $\Psi|_{\mathsf{cd}} = (T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}))|_{\mathsf{cd}}$. This leads to the required result.

case $\mathsf{M}_\nu(\tau_1 \times \tau_2)$: We again have that $v = \nu.\ell$. Therefore, $\Psi(\nu.\ell) = \mathsf{left}(\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$. This implies that $M(\nu.\ell) = \mathsf{inl}\ v'$. We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{inl}\ v' : \mathsf{left}(\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v' : (\mathsf{M}_\nu(\tau_1) \times \mathsf{M}_\nu(\tau_2))$$

This implies that $v' = \langle v_1', v_2' \rangle$. From the typing rules we get that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_1' : \mathsf{M}_\nu(\tau_1)$ and $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v_2' : \mathsf{M}_\nu(\tau_2)$. Applying lemma A.1.24 to the derivations for v_1' and v_2', we get that $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v_1' : \mathsf{M}_\nu(\tau_1)$ and $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_\nu); \cdot; \cdot; \cdot \vdash v_2' : \mathsf{M}_\nu(\tau_2)$. Applying the inductive hypothesis to the derivations for v_1' and v_2' we get that $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash v_1' : \mathsf{C}_{\nu\nu'}(\tau_1)$ and $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash v_2' : \mathsf{C}_{\nu\nu'}(\tau_2)$. From here we can conclude that

$$\begin{array}{l} T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash \\ \quad \mathsf{inl}\ v' : \mathsf{left}(\mathsf{C}_{\nu\nu'}(\tau_1) \times \mathsf{C}_{\nu\nu'}(\tau_2)) \end{array}$$

By the subtyping rule we can conclude that

$$\begin{array}{l} T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_\nu), \nu'; \cdot; \cdot; \cdot \vdash \\ \quad \mathsf{inl}\ v' : (+ \begin{array}{l} \mathsf{left}(\mathsf{C}_{\nu\nu'}(\tau_1) \times \mathsf{C}_{\nu\nu'}(\tau_2)) \\ \mathsf{right}(\mathsf{M}_{\nu'}(\tau_2 \times \tau_2)) \end{array}) \end{array}$$

This is the required result.

case $\mathsf{M}_\nu(\exists t.\tau)$: The proof proceeds exactly as in the previous case. We only need to use the fact that $(\mathsf{C}_{\rho,\rho'}(\tau))[^{\tau'}/_t] = \mathsf{C}_{\rho,\rho'}(\tau[^{\tau'}/_t])$

$\square$

Proposition A.1.27 (Type Preservation)

If $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.

Proof The proof is by cases on the structure of e. For each possible case, we use the typing derivation together with the evaluation step to get the derivation of the new typing judgment. This mostly relies on the substitution lemmas. We only show the more interesting cases.

- $(M, \mathsf{let}\ x = v\ \mathsf{in}\ e) \Longrightarrow (M, e[^v/_x])$

From $\vdash (M, \mathsf{let}\ x = v\ \mathsf{in}\ e)$ we get that $\vdash \overline{M} : \Psi$ and $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e$. The derivation in turns tells us that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \sigma$ and $\Psi; Dom(\Psi); \cdot; \cdot; \cdot, x : \sigma \vdash e$. At this point, we can apply value substitution to get $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash e[{}^{v}/_{x}]$.

- $(M, \mathsf{only}\ \Theta\ \mathsf{in}\ e) \Longrightarrow (M|_{\Theta}, e)$

  Here we get $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{only}\ \Theta\ \mathsf{in}\ e$ which gives us $\Psi|_{\Theta}; \Theta; \cdot; \cdot; \cdot \vdash e$. Since $\overline{M} \subset M$, we have that $\overline{M}|_{\Theta} \subset M|_{\Theta}$. Moreover, we know that $\Theta \subset Dom(\Psi)$. Therefore, $Dom(\Psi|_{\Theta}) = \Theta$.

- $(M, \mathsf{let}\ x = \mathsf{widen}[\nu'][\tau](v)\ \mathsf{in}\ e) \Longrightarrow (M, e[{}^{v}/_{x}])$

  We know that if $\overline{M} : \Psi$, then we have that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{let}\ x = \mathsf{widen}[\nu'][\tau](v)\ \mathsf{in}\ e$. This implies that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v : \mathsf{M}_{\nu}(\tau)$ and $\Psi|_{\mathsf{cd}}; \mathsf{cd}, \nu, \nu'; \cdot; \cdot; , x : \mathsf{C}_{\nu\nu'}(\tau) \vdash e$ for some ν and ν' belonging to $Dom(\Psi)$. By lemma A.1.24 we have that $\Psi|_{\nu,\mathsf{M}}; Dom(\Psi|_{\nu}); \cdot; \cdot; \cdot \vdash v : \mathsf{M}_{\nu}(\tau)$. By lemma A.1.26 we have that $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}); Dom(\Psi|_{\nu}); \cdot; \cdot; \cdot \vdash v : \mathsf{C}_{\nu\nu'}(\tau)$. But we have that $T_{\nu,\nu'}(\Psi|_{\nu,\mathsf{M}}) = \Psi|_{\mathsf{cd}}, \Psi'$ for some Ψ' since the cast leaves the code type unchanged. Also $\Psi' = T_{\nu\nu'}(\Psi|_{\nu,\mathsf{M}})$ Since we can extend the environment, we have that $\Psi|_{\mathsf{cd}}, \Psi'; \mathsf{cd}, \nu, \nu'; \cdot; \cdot; , x : \mathsf{C}_{\nu\nu'}(\tau) \vdash e$. Applying lemma A.1.22.2 we get that $\Psi|_{\mathsf{cd}}, \Psi'; \mathsf{cd}, \nu, \nu'; \cdot; \cdot; \cdot \vdash e[{}^{v}/_{x}]$. This implies that $\Psi|_{\mathsf{cd}}, \Psi', \nu' \mapsto \{\}; \mathsf{cd}, \nu, \nu'; \cdot; \cdot; \cdot \vdash e[{}^{v}/_{x}]$. If $\overline{M} : \Psi|_{\mathsf{cd}}, \Psi', \nu' \mapsto \{\}$, then clearly $\overline{M} \subset M$. This leads to the required result.

$\square$

Lemma A.1.28 (Canonical forms)

1. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v : \mathsf{int}$ *then* $v = n$.

2. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v : \sigma\ \mathsf{at}\ \rho$ *then* $v = \nu.\ell$.

3. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v : \sigma_1 \times \sigma_2$ *then* $v = \langle v_1, v_2 \rangle$.

4. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v : \exists t.\sigma$ *then* $v = \langle t{=}\tau, v'{:}\sigma \rangle$.

5. *If* $\Psi; \Theta'; \cdot; \cdot; \cdot \vdash v : \exists \alpha{:}\Theta.\sigma$ *then* $v = \langle \alpha{:}\Theta{=}\sigma_1, v'{:}\sigma_2 \rangle$.

6. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v : \forall[\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0$ *then* $v = v'[\![\vec{\tau}]\!]$.

7. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v : \forall[t\vec{:}\kappa][\vec{r}](\vec{\sigma}) \rightarrow 0$
   then $v = \lambda[t\vec{:}\kappa][\vec{r}](x\vec{:}\sigma).e$.

Proof The proof follows from the inspection of the typing rules for values. □

Proposition A.1.29 (Progress) *If* $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* (M', e') *such that* $(M, e) \Longrightarrow (M', e')$.

Proof The proof is again by cases on the structure of e. It uses lemma A.1.28. □

A.1.3 Soundness of λ_{GCgen}

In this section, we prove the soundness of λ_{GCgen}. We will prove soundness in the same way – with the subject reduction and progress lemmas. Since λ_{GCgen} is just an extension of λ_{GC}, we will prove the required lemmas only for the new cases. As usual we will use the exchange and widening lemmas for environments throughout the section.

Lemma A.1.30 *If* $\Theta; \Delta; \Phi \vdash \sigma$, *then* $\Theta[^{\nu}/_{r}]; \Delta; \Phi[^{\nu}/_{r}] \vdash \sigma[^{\nu}/_{r}]$.

Proof The proof is by induction over the derivation of σ. □

Lemma A.1.31 $(\Phi[^{\nu}/_{r}])|_{\Theta,\nu} = (\Phi|_{\Theta,r})[^{\nu}/_{r}]$ *and* $(\Gamma[^{\nu}/_{r}])|_{\Theta,\nu} = (\Gamma|_{\Theta,r})[^{\nu}/_{r}]$

Proof Same as lemma A.1.2. □

Lemma A.1.32 $(\mathsf{M}_{\rho_1,\rho_2}(\tau))[^{\tau'}/_{t}] = \mathsf{M}_{\rho_1,\rho_2}(\tau[^{\tau'}/_{t}])$ *and* $(\mathsf{M}_{\rho_1,\rho_2}(\tau))[^{\nu}/_{r}] = \mathsf{M}_{\rho_1[^{\nu}/_{r}],\rho_2[^{\nu}/_{r}]}(\tau)$

Proof Proved by considering the different possible reductions of $\mathsf{M}_{\rho_1,\rho_2}(\tau)$. □

Lemma A.1.33 *If* $\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash op : \sigma$, *then*
$\Psi; \Theta, \nu; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash op[^{\nu}/_{r}] : \sigma[^{\nu}/_{r}]$

Proof The proof is a straightforward induction over the derivation for op. Note that $\Theta, \nu = (\Theta, r)[^{\nu}/_{r}]$. It uses lemma A.1.32 for the case involving subtyping with the $\mathsf{M}_{\rho_1,\rho_2}(\tau)$ type.
case $\langle r' \in \Theta' = \rho, v\rangle$: We have that

$$\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash \langle r' \in \Theta' = \rho, v\rangle : \exists r' \in \Theta'.(\sigma \text{ at } r')$$

This implies that $\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash v : \sigma[^{\rho}/_{r'}]$ at ρ and $\rho \in \Theta'$ and $\Theta' \subset \Theta, r$. Applying the inductive hypothesis we get that

$$\Psi; \Theta, \nu; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash v[^{\nu}/_{r}] : (\sigma[^{\rho}/_{r'}])[^{\nu}/_{r}] \text{ at } \rho[^{\nu}/_{r}]$$

This implies that

$$\Psi;\Theta,\nu;\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}] \vdash$$
$$v[{}^{\nu}/_{r}] \,:\, (\sigma[{}^{\nu}/_{r}])[{}^{(\rho[{}^{\nu}/_{r}])}/_{r'}] \;\mathsf{at}\; \rho[{}^{\nu}/_{r}]$$

Moreover, $\Theta'[{}^{\nu}/_{r}] \subset \Theta,\nu$ and $\rho[{}^{\nu}/_{r}] \in \Theta'[{}^{\nu}/_{r}]$. This implies that

$$\Psi;\Theta,\nu;\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}] \vdash$$
$$\langle r' \in \Theta'[{}^{\nu}/_{r}] = \rho[{}^{\nu}/_{r}], v[{}^{\nu}/_{r}]\rangle \,:\, \exists r' {\in} \{\Theta'[{}^{\nu}/_{r}]\}.(\sigma[{}^{\nu}/_{r}] \;\mathsf{at}\; r')$$

□

Lemma A.1.34 *If* $\Psi;\Theta,r;\Delta;\Phi;\Gamma \vdash e$*, then* $\Psi;\Theta,\nu;\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}] \vdash e[{}^{\nu}/_{r}]$

Proof The proof is by induction over the derivation of e. Note that $\Theta,\nu = (\Theta,r)[{}^{\nu}/_{r}]$. We will consider only the extra cases here.

case $\mathsf{open}\; v \;\mathsf{as}\; \langle r', x\rangle \;\mathsf{in}\; e$: We have that

$$\Psi;\Theta,r;\Delta;\Phi;\Gamma \vdash \mathsf{open}\; v \;\mathsf{as}\; \langle r', x\rangle \;\mathsf{in}\; e$$

This implies that

$$\Psi;\Theta,r,r';\Delta;\Phi;\Gamma,x{:}\sigma \;\mathsf{at}\; r' \vdash e$$

and $\Psi;\Theta,r;\Delta;\Phi;\Gamma \vdash v \,:\, \exists r' {\in} \Theta'.(\sigma \;\mathsf{at}\; r')$. Applying lemma A.1.33 to the derivation for v we get that

$$\Psi;\Theta,\nu;\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}] \vdash v[{}^{\nu}/_{r}] \,:\, \exists r' {\in} \Theta'[{}^{\nu}/_{r}].(\sigma[{}^{\nu}/_{r}] \;\mathsf{at}\; r')$$

Applying the inductive hypothesis to the derivation for e we get that

$$\Psi;\Theta,\nu,r';\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}],x{:}\sigma[{}^{\nu}/_{r}] \;\mathsf{at}\; r' \vdash e[{}^{\nu}/_{r}]$$

From this we get that

$$\Psi; \Theta, \nu; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash \mathsf{open}\ v[^{\nu}/_{r}]\ \mathsf{as}\ \langle r', x\rangle\ \mathsf{in}\ e[^{\nu}/_{r}]$$

case $\mathsf{ifreg}\ (r_1 = r_2)\ e_1\ e_2$: We have that

$$\begin{aligned}&\Psi; (\Theta, r)[^{r', r'}/_{r_1, r_2}]; \Delta; \Phi[^{r', r'}/_{r_1, r_2}]; \Gamma[^{r', r'}/_{r_1, r_2}] \vdash \\ &\quad e_1[^{r', r'}/_{r_1, r_2}]\end{aligned}$$

and $\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash e_2$ and $r' \notin \Theta, r$. Applying the inductive hypothesis to the derivation for e_2 leads to

$$\Psi; \Theta, \nu; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash e_2[^{\nu}/_{r}]$$

Applying the inductive hypothesis to the derivation for e_1 and using the fact that r is different from r_1 and r_2 we get that

$$\begin{aligned}&\Psi; (\Theta, \nu)[^{r', r'}/_{r_1, r_2}]; \Delta; \\ &(\Phi[^{\nu}/_{r}])[^{r', r'}/_{r_1, r_2}]; (\Gamma[^{\nu}/_{r}])[^{r', r'}/_{r_1, r_2}] \vdash \\ &\quad (e_1[^{\nu}/_{r}])[^{r', r'}/_{r_1, r_2}]\end{aligned}$$

This leads to the required result.

case $\mathsf{ifreg}\ (r = r_2)\ e_1\ e_2$: We have that

$$\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash \mathsf{ifreg}\ (r = r_2)\ e_1\ e_2$$

This implies that

$$\begin{aligned}&\Psi; (\Theta, r)[^{r', r'}/_{r, r_2}]; \Delta; \Phi[^{r', r'}/_{r, r_2}]; \Gamma[^{r', r'}/_{r, r_2}] \vdash \\ &\quad e_1[^{r', r'}/_{r, r_2}]\end{aligned}$$

and $\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash e_2$. We must prove that

$$\Psi; \Theta, \nu; \Delta; \Phi[{}^{\nu}/_{r}]; \Gamma[{}^{\nu}/_{r}] \vdash \mathsf{ifreg}\ (\nu = r_2)\ e_1[{}^{\nu}/_{r}]\ e_2[{}^{\nu}/_{r}]$$

This implies that we must prove that $\Psi; \Theta, \nu; \Delta; \Phi[{}^{\nu}/_{r}]; \Gamma[{}^{\nu}/_{r}] \vdash e_2[{}^{\nu}/_{r}]$ and

$$\begin{aligned}&\Psi; (\Theta, \nu)[{}^{\nu}/_{r_2}]; \Delta; (\Phi[{}^{\nu}/_{r}])[{}^{\nu}/_{r_2}]; (\Gamma[{}^{\nu}/_{r}])[{}^{\nu}/_{r_2}] \vdash \\ &\quad (e_1[{}^{\nu}/_{r}])[{}^{\nu}/_{r_2}]\end{aligned}$$

This implies that we must prove that

$$\begin{aligned}&\Psi; (\Theta, r)[{}^{\nu,\nu}/_{r, r_2}]; \Delta; \Phi[{}^{\nu,\nu}/_{r, r_2}]; \Gamma[{}^{\nu,\nu}/_{r, r_2}] \vdash \\ &\quad e_1[{}^{\nu,\nu}/_{r, r_2}]\end{aligned}$$

Applying the inductive hypothesis to the derivation for e_2 leads to the required result for this derivation. In the case for e_1, we substitute for r' and applying the inductive hypothesis and the fact that $r' \notin \Theta, r$ leads to the required result.

case $\mathsf{ifreg}\ (\nu' = r_2)\ e_1\ e_2$: This case follows directly from the inductive hypothesis on the derivations of e_1 and e_2 and using the fact that $\nu' \neq \nu$ and $r \neq r_2$.

case $\mathsf{ifreg}\ (\nu = r_2)\ e_1\ e_2$: This case follows directly from the inductive hypothesis on the derivations of e_1 and e_2 and using the fact that $r \neq r_2$.

case $\mathsf{ifreg}\ (\nu' = r)\ e_1\ e_2$: We have that

$$\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash \mathsf{ifreg}\ (\nu' = r)\ e_1\ e_2$$

This means that we know that

$$\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash e_2$$

We have to prove that

$$\Psi; \Theta, \nu; \Delta; \Phi[{}^{\nu}/_{r}]; \Gamma[{}^{\nu}/_{r}] \vdash \mathsf{ifreg}\ (\nu' = \nu)\ e_1[{}^{\nu}/_{r}]\ e_2[{}^{\nu}/_{r}]$$

This implies that we have to prove that

$$\Psi; \Theta, \nu; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash e_2[^{\nu}/_{r}]$$

Applying the inductive hypothesis to the derivation for e_2 leads to the result.

case $\mathsf{ifreg}\ (\nu = r)\ e_1\ e_2$: We have that

$$\Psi; \Theta, r; \Delta; \Phi; \Gamma \vdash \mathsf{ifreg}\ (\nu = r)\ e_1\ e_2$$

This means that we know that

$$\Psi; \Theta, \nu; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash e_1[^{\nu}/_{r}]$$

We have to prove that

$$\Psi; \Theta, \nu; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash \mathsf{ifreg}\ (\nu = \nu)\ e_1[^{\nu}/_{r}]\ e_2[^{\nu}/_{r}]$$

This implies that we have to prove that

$$\Psi; \Theta, \nu; \Delta; \Phi[^{\nu}/_{r}]; \Gamma[^{\nu}/_{r}] \vdash e_1[^{\nu}/_{r}]$$

But we already know this. □

Lemma A.1.35 *If* $\Delta, t\!:\!\kappa' \vdash \tau : \kappa$ *and* $\Delta \vdash \tau' : \kappa'$, *then* $\Delta \vdash \tau[^{\tau'}/_{t}] : \kappa$

Proof The proof is a straightforward induction over the structure of τ. □

Lemma A.1.36 *If* $\Theta; \Delta, t\!:\!\kappa; \Phi \vdash \sigma$ *and* $\Delta \vdash \tau : \kappa$, *then* $\Theta; \Delta; \Phi \vdash \sigma[^{\tau}/_{t}]$

Proof The proof is again a straighforward induction over the structure of σ. □

Lemma A.1.37 *If* $\Psi; \Theta; \Delta, t\!:\!\kappa; \Phi; \Gamma \vdash op : \sigma$ *and* $\Delta \vdash \tau : \kappa$ *then*
$\Psi; \Theta; \Delta; \Phi; \Gamma[^{\tau}/_{t}] \vdash op[^{\tau}/_{t}] : \sigma[^{\tau}/_{t}]$

Proof The proof is by induction over the derivation for *op*. □

Lemma A.1.38 *If* $\Psi; \Theta; \Delta, t\!:\!\kappa; \Phi; \Gamma \vdash e$ *and* $\cdot \vdash \tau' : \kappa$ *then* $\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\tau'}\!/_{t}] \vdash e[{}^{\tau'}\!/_{t}]$

Proof The proof is again by induction over the derivation for e. □

Lemma A.1.39 *If* $\Theta; \Delta; \Phi, \alpha\!:\!\Theta' \vdash \sigma$ *and* $\Theta'; \Delta; \Phi \vdash \sigma'$, *then* $\Theta; \Delta; \Phi \vdash \sigma[{}^{\sigma'}\!/_{\alpha}]$

Proof The proof is a straighforward induction over the structure of σ. □

Lemma A.1.40 *If* $\Psi; \Theta; \Delta; \Phi, \alpha\!:\!\Theta'; \Gamma \vdash op : \sigma$ *and* $\Theta'; \Delta; \Phi \vdash \sigma'$ *then*
$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\sigma'}\!/_{\alpha}] \vdash op[{}^{\sigma'}\!/_{\alpha}] : \sigma[{}^{\sigma'}\!/_{\alpha}]$

Proof The proof is again by induction over the typing derivation for op. We will consider only the case for region packages.

case $\langle r \in \Theta' = \rho, v\rangle$: We have that

$$\Psi; \Theta; \Delta; \Phi, \alpha\!:\!\Theta'; \Gamma \vdash \langle r \in \Theta' = \rho, v\rangle : \exists r \!\in\! \Theta'.(\sigma \ \mathsf{at}\ r)$$

and $\Theta'; \Delta; \Phi \vdash \sigma'$. This implies that $\Psi; \Theta; \Delta; \Phi, \alpha\!:\!\Theta'; \Gamma \vdash v : \sigma[{}^{\rho}\!/_{r}] \ \mathsf{at}\ \rho$ and $\rho \in \Theta'$ and $\Theta' \subset \Theta$. Applying the inductive hypothesis we get that

$$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\sigma'}\!/_{\alpha}] \vdash v[{}^{\sigma'}\!/_{\alpha}] : (\sigma[{}^{\rho}\!/_{r}])[{}^{\sigma'}\!/_{\alpha}] \ \mathsf{at}\ \rho$$

Since r does not occur in Θ, and $\Theta' \subset \Theta$, therefore r does not occur free in σ'. Therefore we have that

$$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\sigma'}\!/_{\alpha}] \vdash v[{}^{\sigma'}\!/_{\alpha}] : (\sigma[{}^{\sigma'}\!/_{\alpha}])[{}^{\rho}\!/_{r}] \ \mathsf{at}\ \rho$$

This leads to the lemma. □

Lemma A.1.41 *If* $\Psi; \Theta; \Delta; \Phi, \alpha\!:\!\Theta'; \Gamma \vdash e$ *and* $\Theta'; \Delta; \Phi \vdash \sigma'$ *then*
$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\sigma'}\!/_{\alpha}] \vdash e[{}^{\sigma'}\!/_{\alpha}]$

Proof The proof is by induction over the derivation of e.

case $\mathsf{open}\ v\ \mathsf{as}\ \langle r, x\rangle\ \mathsf{in}\ e$: We have that

$$\Psi; \Theta; \Delta; \Phi, \alpha\!:\!\Theta'; \Gamma \vdash v\ :\ \exists r\!\in\!\Theta_1.(\sigma\ \mathsf{at}\ r)$$

and $\Psi; \Theta, r; \Delta; \Phi, \alpha\!:\!\Theta'; \Gamma, x\!:\!\sigma\ \mathsf{at}\ r \vdash e$. Applying lemma A.1.40 to the derivation for v we get that

$$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\sigma'}\!/_{\alpha}] \vdash v[{}^{\sigma'}\!/_{\alpha}]\ :\ \exists r\!\in\!\Theta_1.(\sigma[{}^{\sigma'}\!/_{\alpha}]\ \mathsf{at}\ r)$$

Applying the inductive hypothesis to the derivation for e we get that

$$\Psi; \Theta, r; \Delta; \Phi; \Gamma[{}^{\sigma'}\!/_{\alpha}], x\!:\!\sigma[{}^{\sigma'}\!/_{\alpha}]\ \mathsf{at}\ r \vdash e[{}^{\sigma'}\!/_{\alpha}]$$

This leads to the required result.

case $\mathsf{ifreg}\ (r_1 = r_2)\ e_1\ e_2$: We have that $\Psi; \Theta; \Delta; \Phi, \alpha\!:\!\Theta'; \Gamma \vdash e_2$ and

$$\begin{aligned}&\Psi; \Theta[{}^{r,\,r}\!/_{r_1,\,r_1}]; \Delta;\\ &\Phi[{}^{r,\,r}\!/_{r_1,\,r_1}], \alpha\!:\!\Theta'[{}^{r,\,r}\!/_{r_1,\,r_1}]; \Gamma[{}^{r,\,r}\!/_{r_1,\,r_2}] \vdash\\ &\quad e_1[{}^{r,\,r}\!/_{r_1,\,r_2}]\end{aligned}$$

Applying the inductive hypothesis to the derivation of e_2 leads to

$$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\sigma'}\!/_{\alpha}] \vdash e_2[{}^{\sigma'}\!/_{\alpha}]$$

By lemma A.1.30 we have that

$$\Theta'[{}^{r,\,r}\!/_{r_1,\,r_2}]; \Delta; \Phi[{}^{r,\,r}\!/_{r_1,\,r_2}] \vdash \sigma'[{}^{r,\,r}\!/_{r_1,\,r_2}]$$

Substitute $\sigma'[{}^{r,\,r}/_{r_1,\,r_2}]$ in the derivation for e_1 and by applying the inductive hypothesis we get that

$$\Psi; \Theta[{}^{r,\,r}/_{r_1,\,r_1}]; \Delta; \Phi[{}^{r,\,r}/_{r_1,\,r_1}]; (\Gamma[{}^{\sigma'}/_{\alpha}])[{}^{r,\,r}/_{r_1,\,r_2}] \vdash \\ (e_1[{}^{\sigma'}/_{\alpha}])[{}^{r,\,r}/_{r_1,\,r_2}]$$

This leads to the required result.

case ifreg $(r = \nu)$ e_1 e_2: We have that $\Psi; \Theta; \Delta; \Phi, \alpha : \Theta'; \Gamma \vdash e_2$ and

$$\Psi; \Theta[{}^{\nu}/_{r}]; \Delta; \Phi[{}^{\nu}/_{r}], \alpha : \Theta'[{}^{\nu}/_{r}]; \Gamma[{}^{\nu}/_{r}] \vdash e_1[{}^{\nu}/_{r}]$$

Applying the inductive hypothesis to the derivation of e_2 leads to

$$\Psi; \Theta; \Delta; \Phi; \Gamma[{}^{\sigma'}/_{\alpha}] \vdash e_2[{}^{\sigma'}/_{\alpha}]$$

By lemma A.1.30 we have that

$$\Theta'[{}^{\nu}/_{r}]; \Delta; \Phi[{}^{\nu}/_{r}] \vdash \sigma'[{}^{\nu}/_{r}]$$

Substituting $\sigma'[{}^{\nu}/_{r}]$ in the derivation for e_1 and applying the inductive hypothesis gives us that

$$\Psi; \Theta[{}^{\nu}/_{r}]; \Delta; \Phi[{}^{\nu}/_{r}]; (\Gamma[{}^{\sigma'}/_{\alpha}])[{}^{\nu}/_{r}] \vdash (e_1[{}^{\sigma'}/_{\alpha}])[{}^{\nu}/_{r}]$$

which leads to the required result. □

Lemma A.1.42 *If* $\Psi; \Theta; \Delta; \Phi; \Gamma, x : \sigma' \vdash op : \sigma$ *and* $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v' : \sigma'$ *then* $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash op[{}^{v'}/_{x}] : \sigma$

Proof The proof is by induction over the typing derivation for op. The new cases follow in a straightforward way. □

Lemma A.1.43 *If* $\Psi; \Theta; \Delta; \Phi; \Gamma \vdash v : \sigma$ *and* $\Theta_1; \Delta; \Phi|_{\Theta_1} \vdash \sigma$ *and* $\Theta_1 \subset \Theta$*, then* $\Psi|_{\Theta_1}; \Theta_1; \Delta; \Phi|_{\Theta_1}; \Gamma|_{\Theta_1} \vdash v : \sigma$

Proof The proof is by induction over the derivation for v. We will show only the extra case here.

case $\langle r \in \Theta_2 = \rho, v\rangle$: We have that $\Psi;\Theta;\Delta;\Phi;\Gamma \vdash \langle r \in \Theta_2 = \rho, v\rangle : \exists r \in \Theta_2.(\sigma \ \mathsf{at}\ r)$. This implies that $\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \sigma[{}^{\rho}/_{r}]\ \mathsf{at}\ \rho$ and $\rho \in \Theta_2$ and $\Theta_2 \subset \Theta$. We also have that $\Theta_1;\Delta;\Phi|_{\Theta_1} \vdash \exists r \in \Theta_2.(\sigma\ \mathsf{at}\ r)$. This implies that $\Theta_2 \subset \Theta_1$. Moreover, $\Theta_1, r;\Delta;\Phi|_{\Theta_1} \vdash \sigma$. By lemma A.1.30 we get that $\Theta_1, \rho;\Delta;(\Phi|_{\Theta_1})[{}^{\rho}/_{r}] \vdash \sigma[{}^{\rho}/_{r}]$. But since $\rho \in \Theta_1$ and r does not occur free in $\Phi|_{\Theta_1}$, we get that $\Theta_1;\Delta;\Phi|_{\Theta_1} \vdash \sigma[{}^{\rho}/_{r}]$. Applying the inductive hypothesis to the derivation for v we get that $\Psi|_{\Theta_1};\Theta_1;\Delta;\Phi|_{\Theta_1};\Gamma|_{\Theta_1} \vdash v : \sigma[{}^{\rho}/_{r}]\ \mathsf{at}\ \rho$. This implies that $\Psi|_{\Theta_1};\Theta_1;\Delta;\Phi|_{\Theta_1};\Gamma|_{\Theta_1} \vdash \langle r \in \Theta_2 = \rho, v\rangle : \exists r \in \Theta_2.(\sigma\ \mathsf{at}\ r)$

□

Lemma A.1.44 *If* $\Psi;\Theta;\Delta;\Phi;\Gamma, x{:}\sigma \vdash e$ *and* $\Psi;\Theta;\Delta;\Phi;\Gamma \vdash v : \sigma$ *then* $\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e[{}^{v}/_{x}]$

Proof The proof is by induction over the derivation of e. We will only consider the case for ifreg.

case $\mathsf{ifreg}\ (r = \nu)\ e_1\ e_2$: By definition we have that

$$\Psi;\Theta[{}^{\nu}/_{r}];\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}], x{:}\sigma[{}^{\nu}/_{r}] \vdash e_1[{}^{\nu}/_{r}]$$

and $\Psi;\Theta;\Delta;\Phi;\Gamma, x{:}\sigma \vdash e_2$. Applying the inductive hypothesis to the derivation of e_2, we get that $\Psi;\Theta;\Delta;\Phi;\Gamma \vdash e_2[{}^{v}/_{x}]$. By lemma A.1.33 we know that

$$\Psi;\Theta[{}^{\nu}/_{r}];\Delta;\Phi[{}^{\nu}/_{r}];\Gamma[{}^{\nu}/_{r}] \vdash v[{}^{\nu}/_{r}] : \sigma[{}^{\nu}/_{r}]$$

Substituting $v[{}^{\nu}/_{r}]$ for x in the derivation for e_1 leads to the lemma.

case $\mathsf{ifreg}\ (r_1 = r_2)\ e_1\ e_2$: By definition we have that

$$\begin{array}{l}\Psi;\Theta[{}^{r,r}/_{r_1,r_2}];\Delta;\\ \Phi[{}^{r,r}/_{r_1,r_2}];\Gamma[{}^{r,r}/_{r_1,r_2}], x{:}\sigma[{}^{r,r}/_{r_1,r_2}] \vdash \\ \quad e_1[{}^{r,r}/_{r_1,r_2}]\end{array}$$

and $\Psi; \Theta; \Delta; \Phi; \Gamma, x\!:\!\sigma \vdash e_2$. By lemma A.1.33 we know that

$$\Psi; \Theta[^{r,\,r}/_{r_1,\,r_2}]; \Delta; \Phi[^{r,\,r}/_{r_1,\,r_2}]; \Gamma[^{r,\,r}/_{r_1,\,r_2}] \vdash \\ v[^{r,\,r}/_{r_1,\,r_2}] \,:\, \sigma[^{r,\,r}/_{r_1,\,r_2}]$$

Substituting $v[^{r,\,r}/_{r_1,\,r_2}]$ and v in the derivation of e_1 and e_2 respectively, and applying the inductive hypothesis leads to the lemma. $\Box$

Proposition A.1.45 (Type Preservation) *If* $\vdash (M, e)$ *and* $(M, e) \Longrightarrow (M', e')$ *then* $\vdash (M', e')$.

Proof The proof is by induction over the evaluation relation. We will consider only the additional cases here. The lemma follows in a straightforward way for the ifreg cases.

case $\mathsf{open}\ \langle r \in \Theta = \nu, v\rangle\ \mathsf{as}\ \langle r, x\rangle\ \mathsf{in}\ e$: We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{open}\ \langle r \in \Theta = \nu, v\rangle\ \mathsf{as}\ \langle r, x\rangle\ \mathsf{in}\ e$$

This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \langle r \in \Theta = \nu, v\rangle \,:\, \exists r \!\in\! \Theta.(\sigma\ \mathsf{at}\ r)$$

where $\Theta \subset Dom(\Psi)$. By the typing rule for region packages, we have that $\nu \in Dom(\Psi)$ and that $\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v \,:\, \sigma[^{\nu}/_{r}]\ \mathsf{at}\ \nu$. We also have that $\Psi; Dom(\Psi), r; \cdot; \cdot; x\!:\!\sigma\ \mathsf{at}\ r \vdash e$. By lemma A.1.34 and since $\nu \in Dom(\Psi)$ we get that

$$\Psi; Dom(\Psi); \cdot; \cdot; x\!:\!\sigma[^{\nu}/_{r}]\ \mathsf{at}\ \nu \vdash e[^{\nu}/_{r}]$$

Applying lemma A.1.44 leads to the result. $\Box$

Lemma A.1.46 (Canonical forms)

1. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \,:\, \mathit{int}$ *then* $v = n$.

2. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \,:\, \sigma\ \mathsf{at}\ \nu$ *then* $v = \nu.\ell$.

3. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \,:\, \sigma_1 \times \sigma_2$ *then* $v = \langle v_1, v_2 \rangle$.

4. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \,:\, \exists t\!:\!\kappa.\sigma$ *then* $v = \langle t\!=\!\tau, v'\!:\!\sigma \rangle$.

5. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \,:\, \exists \alpha\!:\!\Theta'.\sigma$ *then* $v = \langle \alpha\!:\!\Theta'\!=\!\sigma_1, v'\!:\!\sigma_2 \rangle$.

6. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \,:\, \exists r\!\in\!\Theta'.(\sigma \text{ at } r)$ *then* $v = \langle r \in \Theta' = \rho, v' \rangle$.

7. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \,:\, \forall [\![\vec{\tau}]\!][\vec{r}](\vec{\sigma}) \xrightarrow{\rho} 0$ *then* $v = v'[\![\vec{\tau}]\!]$.

8. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v \,:\, \forall [t\vec{:}\kappa][\vec{r}](\vec{\sigma}) \to 0$
   then $v = \lambda[t\vec{:}\kappa][\vec{r}](x\vec{:}\sigma).e$.

Proof The proof follows from the inspection of the typing rules for values. □

Proposition A.1.47 (Progress) *If* $\vdash (M, e)$ *then either* $e = \text{halt } v$ *or there exists a* (M', e') *such that* $(M, e) \Longrightarrow (M', e')$.

Proof The proof is by induction over the structure of e. The ifreg cases follow in a straightforward way. We will consider only the region open construct.

case open v as $\langle r, x \rangle$ in e: We know that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \text{open } v \text{ as } \langle r, x \rangle \text{ in } e$$

This means that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash v \,:\, \exists r\!\in\!\Theta.(\sigma \text{ at } r)$$

By lemma A.1.46 we get that $v = \langle r \in \Theta = \rho, v' \rangle$. By the typing rule for region packages, we get that $\rho \in Dom(\Psi)$. This means that $\rho = \nu$ for some ν. Therefore, we have that

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \text{open } \langle r \in \Theta = \nu, v' \rangle \text{ as } \langle r, x \rangle \text{ in } e$$

But by the operational semantics this goes to $(M, e[\nu, v'/r, x])$. □

A.2 Soundness of typed regions

In this section again, I will assume that variable names are unique. It is easy to show by induction that extending Δ, or Γ with additional bindings is safe. Also I will assume that those environments use unique variable names, so there is no duplicate bindings. Finally, the order of bindings in Ψ and Γ is irrelevant.

A.2.1 Soundness of the core language

Lemma A.2.1 (Value Substitution)

If $\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma$ *then:*

- *If* $\Psi; \Delta; \Theta; \Gamma, x{:}\sigma \vdash v' : \sigma'$ *then* $\Psi; \Delta; \Theta; \Gamma \vdash v'[{}^{v}\!/\!_{x}] : \sigma'$.
- *If* $\Psi; \Delta; \Theta; \Gamma, x{:}\sigma \vdash e$ *then* $\Psi; \Delta; \Theta; \Gamma \vdash e[{}^{v}\!/\!_{x}]$.
- *and similarly for all rules using* Γ.

Proof The cases for n, x, λ, and $\nu.\ell$ are immediate, the rest follows by induction on the typing derivation, using the fact that Γ is not ordered. □

Lemma A.2.2 (Type Substitution)

If $\Delta \vdash^{CiC} \varphi : \kappa$ *then:*

- *If* $\Delta, t{:}\kappa \vdash^{CiC} \varphi' : \kappa'$ *then* $\Delta[{}^{\varphi}\!/\!_{t}] \vdash^{CiC} \varphi'[{}^{\varphi}\!/\!_{t}] : \kappa'[{}^{\varphi}\!/\!_{t}]$.
- *If* $\Psi; \Delta, t{:}\kappa; \Theta; \Gamma \vdash v : \sigma$ *then* $\Psi; \Delta[{}^{\varphi}\!/\!_{t}]; \Theta[{}^{\varphi}\!/\!_{t}]; \Gamma[{}^{\varphi}\!/\!_{t}] \vdash v[{}^{\varphi}\!/\!_{t}] : \sigma[{}^{\varphi}\!/\!_{t}]$.
- *If* $\Psi; \Delta, t{:}\kappa; \Theta; \Gamma \vdash e$ *then* $\Psi; \Delta[{}^{\varphi}\!/\!_{t}]; \Theta[{}^{\varphi}\!/\!_{t}]; \Gamma[{}^{\varphi}\!/\!_{t}] \vdash e[{}^{\varphi}\!/\!_{t}]$.
- *and similarly for all rules using* Δ.

Proof The first case is actually a property of CiC.

The cases for n and x are immediate.

The case for λ follows from the fact that Δ, Θ, $\vec{\sigma}$, and e are all closed w.r.t Δ'.

The case for $\nu.\ell$ follows from the fact that Ψ is closed w.r.t type variables, so τ is also necessarily closed.

The rest follows by induction on the typing derivations. □

Lemma A.2.3 (Ψ Weakening)

If $\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma$ *and* $\nu.n \notin Dom(\Psi)$ *and* $\vdash \Psi, \nu.n \mapsto \tau$ *then* $\Psi, \nu.n \mapsto \tau; \Delta; \Theta; \Gamma \vdash v : \sigma$.

Proof By induction on the typing derivation: the only place where Ψ is used is in the rule for $\nu.n$ where weakening is trivially safe. □

Lemma A.2.4 (Θ Irrelevance)

If $\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash v : \sigma$ *then* $\Psi; \Delta; \Theta, \rho \mapsto (\varphi', n'); \Gamma \vdash v : \sigma$.

Proof Trivial since only $Dom(\Theta)$ is actually used in the rules. □

Lemma A.2.5 (Θ Equivalence)

If $\Theta \sim \Theta'$ *and* $\Psi; \Delta; \Theta; \Gamma \vdash e$ *then* $\Psi; \Delta; \Theta'; \Gamma \vdash e$ *and similarly for all judgments using* Θ.

Proof The only possible difference between Θ and Θ' is ordering and induction over the judgments shows that ordering of Θ is not relevant. □

Lemma A.2.6 (Type Preservation)

If $\vdash (M; \Theta; \Psi; e)$ *and* $(M; \Theta; \Psi; e) \Longrightarrow (M'; \Theta'; \Psi'; e')$ *then* $\vdash (M'; \Theta'; \Psi'; e')$.

Proof

Case $\mathsf{let}\ x = v\ \mathsf{in}\ e \Longrightarrow e[{}^{v}/_{x}]$**:**

$$\dfrac{\vdash (M; \Theta; \Psi; \mathsf{let}\ x = v\ \mathsf{in}\ e)}{\dfrac{\Psi; \Theta \vdash M \qquad \bullet \vdash \Theta \qquad \vdash \Psi \qquad \dfrac{\dfrac{\Psi; \bullet; \Theta; \bullet, \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e}{\Psi; \bullet; \Theta; \bullet \vdash v : \sigma \qquad \Psi; \bullet; \Theta; \bullet, x : \sigma \vdash e}\,(\text{inv})}{\Psi; \bullet; \Theta; \bullet \vdash e[{}^{v}/_{x}]}\,(\text{subst})}{\vdash (M; \Theta; \Psi; e[{}^{v}/_{x}])}}\,(\text{inv})$$

Case $\mathsf{let}\ x = \pi_i(v_1, v_2)\ \mathsf{in}\ e \Longrightarrow e[^{v_i}/_{x}]$**:**

As before M, Ψ, and Θ are unchanged, so we just need to show that $\Psi; \bullet; \Theta; \bullet, \vdash \mathsf{let}\ x = \pi_i(v_1, v_2)\ \mathsf{in}\ e$ implies $\Psi; \bullet; \Theta; \bullet \vdash e[^{v_i}/_{x}]$:

$$\dfrac{\dfrac{\Psi; \bullet; \Theta; \bullet, \vdash \mathsf{let}\ x = \pi_i(v_1, v_2)\ \mathsf{in}\ e}{\Psi; \bullet; \Theta; \bullet, x : \sigma_i \vdash e \qquad \dfrac{\dfrac{\Psi; \bullet; \Theta; \bullet \vdash \pi_i(v_1, v_2) : \sigma_i}{\Psi; \bullet; \Theta; \bullet \vdash (v_1, v_2) : \sigma_1 \times \sigma_2}\ (\text{inv})}{\Psi; \bullet; \Theta; \bullet \vdash v_i : \sigma_i}\ (\text{inv})}\ (\text{inv})}{\Psi; \bullet; \Theta; \bullet \vdash e[^{v_i}/_{x}]}\ (\text{subst})$$

Case $\nu.n[\vec{\varphi}](\vec{v}) \Longrightarrow e[^{\vec{\varphi}, \vec{v}}/_{\vec{t}, \vec{x}}]$**:**

Again, M, Θ, and Ψ are unchanged.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Psi; \Theta \vdash M \quad M(\nu.n) = \lambda[\overrightarrow{t : \kappa}]\{\Theta'\}(\overrightarrow{x : \sigma}).e \qquad \dfrac{\dfrac{\Psi; \bullet; \Theta; \bullet \vdash \nu.n[\vec{\varphi}](\vec{v})}{\Psi; \bullet; \Theta; \bullet \vdash \nu.n \mapsto \forall[\overrightarrow{t : \kappa}]\{\Theta'\}(\vec{\sigma}) \to 0}\ (\text{inv})}{\nu \in \mathit{Dom}(\Theta)}\ (\text{inv})}{\Psi; \bullet; \bullet; \bullet \vdash \lambda[\overrightarrow{t : \kappa}]\{\Theta'\}(\overrightarrow{x : \sigma}).e : \forall[\overrightarrow{t : \kappa}]\{\Theta'\}(\vec{\sigma}) \to 0}\ (\text{inv})}{\dfrac{\overrightarrow{t : \kappa} \vdash \Theta'}{\bullet \vdash \Theta'[^{\vec{\varphi}}/_{\vec{t}}]} \qquad \dfrac{\Psi; \overrightarrow{t : \kappa}; \Theta'; \overrightarrow{x : \sigma} \vdash e}{\Psi; \bullet; \Theta'[^{\vec{\varphi}}/_{\vec{t}}]; \overrightarrow{x : \sigma}[^{\vec{\varphi}}/_{\vec{t}}] \vdash e[^{\vec{\varphi}}/_{\vec{t}}]}\ (\text{type subst})}\ (\text{inv})}{\Psi; \bullet; \Theta; \overrightarrow{x : \sigma}[^{\vec{\varphi}}/_{\vec{t}}] \vdash e[^{\vec{\varphi}}/_{\vec{t}}]}\ (\Theta\ \text{equiv})}{\Psi; \bullet; \Theta; \bullet \vdash e[^{\vec{\varphi}, \vec{v}}/_{\vec{t}, \vec{x}}]}\ (\text{value subst})$$

The two type substitution steps rely implicitly on the fact that $\bullet \vdash \vec{\varphi} : \overrightarrow{t : \kappa}$ which we get by inversion from $\Psi; \bullet; \Theta; \bullet \vdash \nu.n[\vec{\varphi}](\vec{v})$. This same inversion gives us $\Theta \sim \Theta'[^{\vec{\varphi}}/_{\vec{t}}]$ which is implicitly used in the Θ equivalence step and $\Psi; \bullet; \Theta; \bullet \vdash v_i : \sigma_i[^{\vec{\varphi}}/_{\vec{t}}]$, also used implicitly in the value substitution.

Case $\mathsf{let}\ x = \mathsf{get}\ \nu.n\ \mathsf{in}\ e \Longrightarrow e[{}^{M(\nu.n)}/_{x}]$**:**

Again M, Ψ, and Θ stay unchanged, so we only need to show:

$$\dfrac{\Psi;\bullet;\Theta;\bullet \vdash \mathsf{let}\ x = \mathsf{get}\ \nu.n\ \mathsf{in}\ e}{\dfrac{\Psi;\bullet;\Theta;\bullet, x{:}\varphi\ n\ \tau \vdash e \qquad \dfrac{\Theta(\nu) = (\varphi, m) \qquad \dfrac{\dfrac{\dfrac{\Psi;\bullet;\Theta;\bullet \vdash \mathsf{get}\ \nu.n : \varphi\ n\ \tau}{\Psi;\bullet;\Theta;\bullet \vdash \nu.n \mapsto \varphi\ n\ \tau}\ (\text{inv})}{\Psi;\bullet;\Theta;\bullet \vdash \nu.n : \tau\ \mathsf{at}\ \nu.n}\ (\text{inv})}{\tau = \Psi(\nu.n)}\ (\text{inv})}{\Psi;\bullet;\Theta;\bullet \vdash M(\nu.n) : \varphi\ n\ \tau}\ (\Psi;\Theta \vdash M)}{\Psi;\bullet;\Theta;\bullet \vdash e[{}^{M(\nu.n)}/_{x}]}\ (\text{subst})}\ (\text{inv})$$

Case $\mathsf{let}\ x = \mathsf{put}[\nu, \tau]\ v\ \mathsf{in}\ e \Longrightarrow e[{}^{\nu.n}/_{x}]$**:**

This time M, Ψ, and Θ change. So we need to show well-formedness of environments. The case of $\Theta, \nu \mapsto (\varphi, n{+}1)$ is trivial since the well-formedness only requires a natural number but does not care about its value. The proof for $\Psi, \nu.n \mapsto \tau$ is:

$$\dfrac{\vdash \Psi \qquad \dfrac{\Psi;\bullet;\Theta,\nu \mapsto (\varphi, n);\bullet \vdash \mathsf{let}\ x = \mathsf{put}[\nu,\tau]\ v\ \mathsf{in}\ e}{\bullet \vdash^{CiC} \nu : \mathsf{R}\ \kappa \qquad \bullet \vdash^{CiC} \tau : \kappa}\ (\text{inv})}{\vdash \Psi, \nu.n \mapsto \tau}$$

The proof for $M, \nu.n \mapsto v$ is:

$$\dfrac{\Psi;\Theta,\nu \mapsto (\varphi, n) \vdash M \qquad \dfrac{\dfrac{\Psi;\bullet;\Theta,\nu \mapsto (\varphi, n);\bullet \vdash \mathsf{let}\ x = \mathsf{put}[\nu,\tau]\ v\ \mathsf{in}\ e}{\Psi;\bullet;\Theta;\bullet \vdash v : \varphi\ n\ \tau}\ (\text{inv})}{\Psi, \nu.n \mapsto \tau;\bullet;\Theta;\bullet \vdash v : \varphi\ n\ \tau}\ (\Psi\ \text{weakening})}{\Psi, \nu.n \mapsto \tau;\Theta,\nu \mapsto (\varphi, n{+}1) \vdash M, \nu.n \mapsto v}$$

The weakening step is justified by the fact that $\vdash \Psi, \nu.n \mapsto \tau$ and $\Psi;\Theta,\nu \mapsto (\varphi, n) \vdash M$ implies $\nu.n \notin \mathit{Dom}(\Psi)$ by inversion.

We still need to show the well-formedness of $e[{}^{\nu.n}/_{x}]$:

$$\dfrac{\dfrac{\dfrac{\Psi;\bullet;\Theta,\nu \mapsto (\varphi, n);\bullet \vdash \mathsf{let}\ x = \mathsf{put}[\nu,\tau]\ v\ \mathsf{in}\ e}{\Psi;\bullet;\Theta,\nu \mapsto (\varphi, n{+}1);\bullet, x{:}\tau\ \mathsf{at}\ \nu.n \vdash e}\ (\text{inv})}{\Psi, \nu.n \mapsto \tau;\bullet;\Theta,\nu \mapsto (\varphi, n{+}1);\bullet, x{:}\tau\ \mathsf{at}\ \nu.n \vdash e}\ (\Psi\ \text{weakening})}{\Psi, \nu.n \mapsto \tau;\bullet;\Theta,\nu \mapsto (\varphi, n{+}1);\bullet \vdash e[{}^{\nu.n}/_{x}]}\ (\text{subst})$$

The weakening step is justified as above. The substitution step relies on the fact that $\Psi, \nu.n \mapsto \tau; \bullet; \Theta, \nu \mapsto (\varphi, n{+}1); \bullet \vdash \nu.n : \tau$ at $\nu.n$ which follows from the typing rule of $\nu.n$.

Case $\mathsf{freergn}\ \nu; e \Longrightarrow e$**:**

Since M, Ψ, and Θ change we need to show their well-formedness. For Ψ and Θ this is trivial since their elements are independent. For M, we have that $\Delta \vdash \Theta, \nu \mapsto (\varphi, n)$ implies that $\nu \notin \mathit{Dom}(\Theta)$, so $\Psi \backslash \nu; \Theta \vdash M \backslash \nu$ derives straightforwardly from $\Psi; \Theta, \nu \mapsto (\varphi, n) \vdash M$. The proof for the well-formedness of the term is:

$$\dfrac{\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash \mathsf{freergn}\ \nu; e}{\Psi; \bullet; \Theta; \bullet \vdash e}\ (\text{inv})}{\Psi \backslash \nu; \bullet; \Theta; \bullet \vdash e}\ (\nu \notin \mathit{Dom}(\Theta))$$

where the last step is justified by the fact that Ψ is only used in the typing rule for $\nu.n$ and that only if $\nu \in \mathit{Dom}(\Theta)$.

Case $\mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e \Longrightarrow e[{}^{\nu}/_{r}]$**:**

Since Θ changes, so we need to show well-formedness of Θ and M. For M it's easy since the new region has size zero, so we only need to show $\nu.j \notin \mathit{Dom}(\Psi)$ which follows from the fact that ν is a brand new unique value. For Θ the proof is:

$$\dfrac{\bullet \vdash \Theta \quad \nu \notin \mathit{Dom}(\Theta) \quad \bullet \vdash^{CiC} \nu : \mathsf{R}\ \kappa \quad \bullet \vdash^{CiC} 0 : \mathsf{Nat} \quad \dfrac{\Psi; \bullet; \Theta; \bullet \vdash \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e}{\bullet \vdash^{CiC} \varphi : \mathsf{Nat} \to \kappa \to \Omega}}{\bullet \vdash \Theta, \nu \mapsto (\varphi, 0)}$$

The proof for the well-formedness of $e[{}^{\nu}/_{r}]$ is:

$$\dfrac{\bullet \vdash^{CiC} \nu : \mathsf{R}\ \kappa \quad \dfrac{\Psi; \bullet; \Theta; \bullet \vdash \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e}{\Psi; \bullet, r : \mathsf{R}\ \kappa; \Theta, r \mapsto (\varphi, 0); \bullet \vdash e}\ (\text{inv})}{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, 0); \bullet \vdash e[{}^{\nu}/_{r}]}\ (\text{subst})$$

Case $\mathsf{cast}[P]\ \nu \mapsto \varphi; e \Longrightarrow e$**:**

Since Θ changes, we need to show well-formedness of M and Θ. For Θ, we first need to show that the new type of the region has the proper kind. I could have added it to the typing rule as well, but it can be inferred:

$$\dfrac{\dfrac{\bullet \vdash \Theta, \nu \mapsto (\varphi, n)}{\bullet \vdash^{CiC} \varphi : \mathsf{Nat} \to \kappa \to \Omega}\ (\text{inv}) \quad \dfrac{\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash \mathsf{cast}[P]\ \nu \mapsto \varphi'; e}{\bullet \vdash^{CiC} P : \Pi i\!:\!0..n{-}1.\, \Pi t\!:\!\kappa.\, (\varphi\ i\ t) = (\varphi'\ i\ t)}\ (\text{inv})}{\mathsf{kindof}(\varphi) = \mathsf{kindof}(\varphi')}\ (\text{CiC})}{\bullet \vdash^{CiC} \varphi' : \mathsf{Nat} \to \kappa \to \Omega}$$

The well-formedness of Θ is then immediate:

$$\dfrac{\bullet \vdash^{CiC} \varphi' : \mathsf{Nat} \to \kappa \to \Omega \qquad \dfrac{\bullet \vdash \Theta, \nu \mapsto (\varphi, n)}{\bullet \vdash \Theta \quad r \notin \mathit{Dom}(\Theta) \quad \Delta \vdash^{CiC} \nu : \mathsf{R}\ \kappa \quad \Delta \vdash^{CiC} n : \mathsf{Nat}}}{\bullet \vdash \Theta, \nu \mapsto (\varphi', n)}$$

The well-formedness of M is trivially preserved for regions other than ν, so we only need to show preservation of M for ν. Since the size does not change and Ψ does not change, the part of the rule that checks $\nu.j \notin \mathit{Dom}(\Psi)$ when $j \geq n$ is also trivially preserved and the rest follows:

$$\dfrac{\dfrac{\forall j\!:\!0..n{-}1\ .\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash M(\nu.j) : \varphi\ j\ (\Psi(\nu.j))}{\forall j\!:\!0..n{-}1\ .\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash M(\nu.j) : \varphi'\ j\ (\Psi(\nu.j))}\ (\forall t\!:\!\kappa.(\varphi'\ j\ t) = (\varphi\ j\ t))}{\forall j\!:\!0..n{-}1\ .\Psi; \bullet; \Theta, \nu \mapsto (\varphi', n); \bullet \vdash M(\nu.j) : \varphi'\ j\ (\Psi(\nu.j))}\ (\Theta\ \text{irrelevance})$$

Finally, the well-formedness of e is trivial:

$$\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi', n); \bullet \vdash \mathsf{cast}[P]\ \nu \mapsto \varphi; e}{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash e}\ (\text{inv})$$

Case $\mathsf{set}\ \nu.n \stackrel{\varphi}{:=} v; e \Longrightarrow e$**:**

Since M and Θ change, we need to show their well-formedness. For Θ all that changes is the φ' type of the region ν, so we only need to show that the new type has the proper kind:

$$\dfrac{\bullet \vdash^{CiC} \varphi' : \mathsf{Nat} \to \kappa \to \Omega \qquad \dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi', n'); \bullet \vdash \mathsf{set}\ \nu.n \stackrel{\varphi}{:=} v; e}{\bullet \vdash^{CiC} \varphi : \kappa \to \Omega}\ (\text{inv})}{\bullet \vdash^{CiC} upd\ \varphi'\ n\ \varphi : \mathsf{Nat} \to \kappa \to \Omega}$$

The proof for M is decomposed into the part that changed (i.e. the location $\nu.n$):

$$\dfrac{\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi', n'); \bullet \vdash \mathsf{set}\ \nu.n \stackrel{\varphi}{:=} v; e}{\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi', n'); \bullet \vdash v : \varphi\ \tau}{\Psi; \bullet; \Theta, \nu \mapsto (upd\ \varphi'\ n\ \varphi, n'); \bullet \vdash v : \varphi\ \tau} \qquad \dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi', n'); \bullet \vdash \nu.n : \tau\ \mathsf{at}\ \nu.n}{\tau = \Psi(\nu.n)}}\ (\text{inv})}{\Psi; \bullet; \Theta, \nu \mapsto (upd\ \varphi'\ n\ \varphi, n'); \bullet \vdash v : (upd\ \varphi'\ n\ \varphi)\ n\ (\Psi(\nu.n))}$$

where the two middle steps use Θ irrelevance and inversion respectively. And the rest follows from Θ irrelevance:

$$\dfrac{j \neq n \quad \Psi; \bullet; \Theta, \nu \mapsto (\varphi', n'); \bullet \vdash M(\nu.j) : \varphi'\ j\ (\Psi(\nu.j))}{\Psi; \bullet; \Theta, \nu \mapsto (upd\ \varphi'\ n\ \varphi, n'); \bullet \vdash M(\nu.j) : (upd\ \varphi'\ n\ \varphi)\ j\ (\Psi(\nu.j))}$$

Finally, the term well-formedness follows immediately by inversion of the typing rule:

$$\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi', n'); \bullet \vdash \mathsf{set}\ \nu.n \stackrel{\varphi}{:=} v; e}{\Psi; \bullet; \Theta, \nu \mapsto (upd\ \varphi'\ n\ \varphi, n'); \bullet \vdash e}\ (\text{inv})$$

□

Lemma A.2.7 (Progress)

If $\vdash (M; \Theta; \Psi; e)$*, either* $e = \mathsf{halt}\ v$ *or* $(M; \Theta; \Psi; e) \Longrightarrow (M'; \Theta'; \Psi'; e')$*.*

Proof By case analysis: if $\vdash (M; \Theta; \Psi; e)$ then the type derivation of e can be one of the following:

Case $\mathsf{halt}\ v$**:**

Trivial.

Case $\mathsf{let}\ x = op\ \mathsf{in}\ e$**:**

This case is subdivided into the next three subcases since op can be either a value v or a projection $\pi_i v$ or a memory access $\mathsf{get}\ v$.

Case $\mathsf{let}\ x = v\ \mathsf{in}\ e$**:**

This case is trivial: $(M; \Theta; \Psi; \mathsf{let}\ x = v\ \mathsf{in}\ e) \Longrightarrow (M; \Theta; \Psi; e[^{v}/_{x}])$

Case $\mathsf{let}\ x = \pi_i v\ \mathsf{in}\ e$**:**

$$\cfrac{\cfrac{\cfrac{\Psi; \bullet; \Theta; \bullet \vdash \mathsf{let}\ x = \pi_i v\ \mathsf{in}\ e}{\Psi; \bullet; \Theta; \bullet \vdash \pi_i v : \sigma}\ \text{(inv)}}{\Psi; \bullet; \Theta; \bullet \vdash v : \sigma_1 \times \sigma_2}\ \text{(inv)}}{v = (v_1, v_2)}\ \text{(inv knowing that } \Gamma = \bullet)$$

So we can reduce with: $(M; \Theta; \Psi; \mathsf{let}\ x = \pi_i(v_1, v_2)\ \mathsf{in}\ e) \Longrightarrow (M; \Theta; \Psi; e[^{v_i}/_{x}])$

Case $\mathsf{let}\ x = \mathsf{get}\ v\ \mathsf{in}\ e$**:**

$$\cfrac{\cfrac{\cfrac{\cfrac{\Psi; \bullet; \Theta; \bullet \vdash \mathsf{let}\ x = \mathsf{get}\ v\ \mathsf{in}\ e}{\Psi; \bullet; \Theta; \bullet \vdash \mathsf{get}\ v : \sigma}\ \text{(inv)}}{\Psi; \bullet; \Theta; \bullet \vdash v \mapsto \sigma}\ \text{(inv)}}{\Psi; \bullet; \Theta; \bullet \vdash v : \tau\ \mathsf{at}\ \rho.n \qquad \Theta(\rho) = (\varphi, m)}\ \text{(inv)}}{v = \nu.n \qquad \Psi(\nu.n) = \tau}\ \text{(inv knowing that } \Gamma = \bullet)$$

Combining the above with $\Psi; \Theta \vdash M$, we can derive that $M(\nu.n)$ exists (and has type σ), so we can reduce with: $(M; \Theta; \Psi; \mathsf{let}\ x = \mathsf{get}\ \nu.n\ \mathsf{in}\ e) \Longrightarrow (M; \Theta; \Psi; e[^{M(\nu.n)}/_{x}])$

Case $v[\vec{\varphi}](\vec{v})$**:**

$$\cfrac{\cfrac{\cfrac{\Psi; \bullet; \Theta; \bullet \vdash v[\vec{\varphi}](\vec{v})}{\Psi; \bullet; \Theta; \bullet \vdash v \mapsto \forall[\overrightarrow{t:\kappa}]\{\Theta'\}(\vec{\sigma}) \to 0}\ \text{(inv)}}{\varphi\ n\ \tau = \forall[\overrightarrow{t:\kappa}]\{\Theta'\}(\vec{\sigma}) \to 0 \qquad \cfrac{\Psi; \bullet; \Theta; \bullet \vdash v : \tau\ \mathsf{at}\ \rho.n \quad \Theta(\rho) = (\varphi, m)}{\rho = \nu \qquad v = \nu.n \qquad \tau = \Psi(\nu.n)}\ \text{(inv \& } \Gamma = \bullet)}\ \text{(inv)}}{\varphi\ n\ (\Psi(\nu.n)) = \forall[\overrightarrow{t:\kappa}]\{\Theta'\}(\vec{\sigma}) \to 0 \qquad \Theta(\nu) = (\varphi, m)}$$

Combining the above with $\Psi;\Theta \vdash M$ we get:

$$\cfrac{\Psi;\Theta \vdash M \quad \Theta(\nu) = (\varphi, m) \quad \varphi\ n\ (\Psi(\nu.n)) = \forall[\overrightarrow{t:\kappa}]\{\Theta'\}(\vec{\sigma}) \to 0}{\cfrac{\Psi;\bullet;\Theta;\bullet \vdash M(\nu.n) : \forall[\overrightarrow{t:\kappa}]\{\Theta'\}(\vec{\sigma}) \to 0}{M(\nu.n) = \lambda[\overrightarrow{t:\kappa}]\{\Theta'\}(\overrightarrow{x:\sigma}).e}\ (\text{inv knowing that } \Gamma = \bullet)}$$

So we can reduce with: $(M;\Theta;\Psi;\nu.n[\vec{\varphi}](\vec{v})) \Longrightarrow (M;\Theta;\Psi;e[{}^{\vec{\varphi},\vec{v}}/_{\vec{t},\vec{x}}])$

Case $\mathsf{let}\ x = \mathsf{put}[\rho,\tau]\ v\ \mathsf{in}\ e$**:**

$$\cfrac{\bullet \vdash \Theta \quad \cfrac{\Psi;\bullet;\Theta;\bullet \vdash \mathsf{let}\ x = \mathsf{put}[\rho,\tau]\ v\ \mathsf{in}\ e}{\Theta = \Theta', \rho \mapsto (\varphi, n)}\ (\text{inv})}{\cfrac{\bullet \vdash^{CiC} \rho : \mathsf{R}\ \kappa}{\rho = \nu}\ (\text{inv with } \Delta = \bullet)}\ (\text{inv})$$

So we can reduce with:

$(M;\Theta;\Psi;\mathsf{let}\ x = \mathsf{put}[\nu,\tau]\ v\ \mathsf{in}\ e) \Longrightarrow (M, \nu.n \mapsto v; \Theta', \nu \mapsto (\varphi, n{+}1); \Psi, \nu.n \mapsto \tau; e[{}^{\nu.n}/_{x}])$

Case $\mathsf{freergn}\ \rho; e$**:**

The proof for this case is actually the same as the proof above for put. We end up with $\rho = \nu$ and can thus reduce with: $(M;\Theta;\Psi;\mathsf{freergn}\ \nu; e) \Longrightarrow (M\backslash\nu;\Theta';\Psi\backslash\nu;e)$

Case $\mathsf{cast}[P]\ \rho \mapsto \varphi; e$**:**

Here the proof is also the same as above, and we reduce with:

$(M;\Theta;\Psi;\mathsf{cast}[P]\ \nu \mapsto \varphi; e) \Longrightarrow (M;\Theta', \nu \mapsto (\varphi, n);\Psi;e)$

Case $\mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e$**:**

$$\cfrac{\Psi;\bullet;\Theta;\bullet \vdash \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e}{\bullet \vdash^{CiC} \varphi : \mathsf{Nat} \to \kappa \to \Omega}\ (\text{inv})$$

So we can reduce with: $(M;\Theta;\Psi;\mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e) \Longrightarrow (M;\Theta, \nu \mapsto (\varphi, 0);\Psi;e[{}^{\nu}/_{r}])$

Case $\mathsf{set}\ v \stackrel{\varphi}{:=} v'; e$**:**

$$\dfrac{\Psi; \bullet; \Theta; \bullet \vdash \mathsf{set}\ v \stackrel{\varphi}{:=} v'; e}{\Theta = \Theta', \rho \mapsto (\varphi, n) \qquad \dfrac{\Psi; \bullet; \Theta; \bullet \vdash v : \tau\ \mathsf{at}\ \rho.m}{\rho = \nu \qquad v = \nu.m}\ (\text{inv with } \Gamma = \bullet)}\ (\text{inv})$$

So we can reduce with:

$(M; \Theta, \nu \mapsto (\varphi', n'); \Psi; \mathsf{set}\ \nu.n \stackrel{\varphi}{:=} v; e) \Longrightarrow (M, \nu.n \mapsto v; \Theta, \nu \mapsto (upd\ \varphi'\ n\ \varphi, n'); \Psi; e)$ $\square$

A.2.2 Soundness of the full language

The statements and proofs of Ψ Weakening, Value Substitution, and Type Substitution lemmas are the same as they were for the core language, so I will not restate them.

Lemma A.2.8 (Θ irrelevance)

If $\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash v : \sigma$ *then* $\Psi; \Delta; \Theta, \rho \mapsto (\varphi', n'); \Gamma \vdash v : \sigma$.

Proof Simple proof by induction on the type derivation of v. There are only two cases where Θ is used: First, in the typing of $\nu.n$ where only $Dom(\Theta)$ is used, so the proof is immediate. Second, in the typing of $\langle \rho, v \rangle$, where we need to show that $\Theta \vdash \rho \in \vec{\rho}$ implies $\Theta' \vdash \rho \in \vec{\rho}$, which is also immediate because this judgment only depends on region aliases and not region types. □

Lemma A.2.9 (Θ equivalence)

If $\Theta' \lhd \Theta$ *then:*

- $\Theta \vdash \rho \in \vec{\rho}$ *implies* $\Theta' \vdash \rho \in \vec{\rho}$.
- $\Theta \lhd \Theta''$ *implies* $\Theta' \lhd \Theta''$.
- $\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma$ *implies* $\Psi; \Delta; \Theta'; \Gamma \vdash v : \sigma$.
- $\Theta \vdash \tau \;\mathsf{at}\; \rho.n \mapsto \sigma$ *implies* $\Theta' \vdash \tau \;\mathsf{at}\; \rho.n \mapsto \sigma$.
- $\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \sigma$ *implies* $\Psi; \Delta; \Theta'; \Gamma \vdash v \mapsto \sigma$.
- $\Psi; \Delta; \Theta; \Gamma \vdash op : \sigma$ *implies* $\Psi; \Delta; \Theta'; \Gamma \vdash op : \sigma$.
- $\Psi; \Delta; \Theta, \Theta''; \Gamma \vdash e$ *implies* $\Psi; \Delta; \Theta,' \Theta''; \Gamma \vdash e$.

Proof The equivalence relation of the full language is more complex than the one of the core language, so the proof is not as immediate. But it is still fairly simple since the only thing that can change is ordering, which we have seen to be unimportant, and region aliases.

All proofs are done by induction over the main derivation, except for the proof for $\Theta \vdash \rho \in \vec{\rho}$ which is done by induction over the derivation of $\Theta' \triangleleft \Theta$. All cases of this first proof follow immediately by induction or by inversion.

The only moderately interesting cases of the proof for $\Theta \triangleleft \Theta''$ are the ones for region aliases:

$$\dfrac{\dfrac{\Theta, \rho \mapsto \vec{\rho} \triangleleft \Theta''}{\Theta \triangleleft \Theta''} \quad \dfrac{\Theta' \triangleleft \Theta, \rho \mapsto \vec{\rho}}{\Theta' \triangleleft \Theta}}{\Theta' \triangleleft \Theta''} \qquad \text{and} \qquad \dfrac{\dfrac{\Theta' \triangleleft \Theta \quad \dfrac{\Theta \triangleleft \Theta'', \rho \mapsto \vec{\rho}}{\Theta \triangleleft \Theta'' \qquad \Theta \vdash \rho \in \vec{\rho}}\,\text{(inv)}}{\Theta' \triangleleft \Theta'' \qquad \Theta' \vdash \rho \in \vec{\rho}}\,\text{(ind)}}{\Theta' \triangleleft \Theta'', \rho \mapsto \vec{\rho}}$$

For the proof for $v : \sigma$, the only case where Θ is used is for $\nu.n$ where one only needs to show that $\Theta(\nu) = (\varphi, m)$ implies $\Theta'(\nu) = (\varphi, m)$.

The rest of the proofs follows directly by induction. $\square$

Lemma A.2.10 (Subsumption)

If $\Delta \vdash \sigma \triangleleft \sigma'$ *and* $\Delta \vdash^{CiC} \sigma : \Omega$ *then* $\Delta \vdash^{CiC} \sigma' : \Omega$.

If $\Delta \vdash \sigma \triangleleft \sigma'$ *and* $\Psi; \Delta; \Theta; \bullet \vdash v : \sigma$ *then* $\Psi; \Delta; \Theta; \bullet \vdash v : \sigma'$.

Proof The first part is by induction on the derivation of $\sigma \triangleleft \sigma'$. The second is by induction over the derivation of $v : \sigma$. The interesting case is for the existential. Inverting $\Delta \vdash \exists r \in \vec{\rho}.\sigma \triangleleft \exists r \in \vec{\rho}'.\sigma'$ gives us $\vec{\rho} \subset \vec{\rho}'$, and $\Delta, r : \mathsf{R}\ \kappa \vdash \sigma \triangleleft \sigma'$, and $\Delta \vdash^{CiC} \rho'_i : \mathsf{R}\ \kappa$. Inverting $\Psi; \Delta; \Theta; \bullet \vdash \langle \rho, v \rangle : \exists r \in \vec{\rho}.\sigma$ gives us $\Theta \vdash \rho \in \vec{\rho}$, and $\Psi; \Delta; \Theta; \bullet \vdash v : \sigma[{}^{\rho}/_{r}]$, and $\Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa$, and $\Delta, r : \mathsf{R}\ \kappa \vdash^{CiC} \sigma : \Omega$. The interesting part is:

$$\dfrac{\dfrac{\vec{\rho} \subset \vec{\rho}' \quad \Theta \vdash \rho \in \vec{\rho}}{\Theta \vdash \rho \in \vec{\rho}'} \quad \dfrac{\dfrac{\Delta, r : \mathsf{R}\ \kappa \vdash \sigma \triangleleft \sigma' \quad \Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa}{\Delta \vdash \sigma[{}^{\rho}/_{r}] \triangleleft \sigma'[{}^{\rho}/_{r}]} \qquad \Psi; \Delta; \Theta; \bullet \vdash v : \sigma[{}^{\rho}/_{r}]}{\Psi; \Delta; \Theta; \bullet \vdash v : \sigma'[{}^{\rho}/_{r}]}}{\Psi; \Delta; \Theta; \Gamma \vdash \langle \rho, v \rangle : \exists r \in \vec{\rho}'.\sigma'}$$

The inference at the extreme left is actually a small lemma that we need to show separately via trivial induction over the derivation of $\Theta \vdash \rho \in \vec{\rho}$. $\square$

Lemma A.2.11 (Type Preservation)

If $\vdash (M;\Theta;\Psi;e)$ *and* $(M;\Theta;\Psi;e) \Longrightarrow (M';\Theta';\Psi';e')$ *then* $\vdash (M';\Theta';\Psi';e')$.

Proof The typing and reduction rules for the terms that were already in the core language are the same and their proof is also the same except of course that they use the new lemmas above. For cast the rule has been slightly modified to use $\lhd$ rather than equality, but the proof is not affected. None of the new cases modify the environment, so we only need to show the well-formedness of the term:

Case $\mathsf{let}\ x = \mathsf{cast}[P]\ v\ \mathsf{in}\ e \Longrightarrow e[^v/_x]$**:**

$$\dfrac{\dfrac{\Psi;\bullet;\Theta;\bullet \vdash \mathsf{let}\ x = \mathsf{cast}[P]\ v\ \mathsf{in}\ e}{\Psi;\bullet;\Theta;\bullet,x{:}\sigma \vdash e \qquad \dfrac{\dfrac{\Psi;\bullet;\Theta;\bullet \vdash \mathsf{cast}[P]\ v : \sigma}{\dfrac{\bullet \vdash^{CiC} P : \sigma' \lhd \sigma}{\Psi;\bullet;\Theta;\bullet \vdash v : \sigma' \qquad \bullet \vdash \sigma' \lhd \sigma}}\,(\text{inv})}{\Psi;\bullet;\Theta;\bullet \vdash v : \sigma}\,(\text{subsumption})}\,(\text{inv})}{\Psi;\bullet;\Theta;\bullet \vdash e[^v/_x]}\,(\text{subst})$$

Case $\mathsf{let}\ \langle t,x\rangle = \mathsf{open}\ \langle \varphi,v\rangle\ \mathsf{in}\ e \Longrightarrow e[^{\varphi,\,v}/_{t,\,x}]$**:**

$$\begin{array}{c}
\Psi;\bullet;\Theta;\bullet \vdash \mathsf{let}\ \langle t,x\rangle = \mathsf{open}\ \langle \varphi,v\rangle\ \mathsf{in}\ e \\ \hline
\Psi;\bullet,t{:}\kappa;\Theta;\bullet,x{:}\sigma[^t/_{t'}] \vdash e \qquad \dfrac{\Psi;\bullet;\Theta;\bullet \vdash \langle\varphi,v\rangle : \exists t'.\kappa\sigma}{\bullet \vdash^{CiC} \varphi : \kappa \qquad \Psi;\bullet;\Theta;\bullet \vdash v : \sigma[^\varphi/_{t'}]}\,(\text{inv}) \quad (\text{inv}) \\ \hline
\Psi;\bullet;\Theta[^\varphi/_{t'}];\bullet,x{:}\sigma[^\varphi/_{t'}] \vdash e[^\varphi/_t] \qquad \Psi;\bullet;\Theta;\bullet \vdash v : \sigma[^\varphi/_{t'}] \quad (\text{subst}) \\ \hline
\Psi;\bullet;\Theta;\bullet,x{:}\sigma[^\varphi/_{t'}] \vdash e[^\varphi/_t] \qquad \Psi;\bullet;\Theta;\bullet \vdash v : \sigma[^\varphi/_{t'}] \quad (t' \text{ free in } \Theta) \\ \hline
\Psi;\bullet;\Theta;\bullet \vdash e[^{\varphi,\,v}/_{t,\,x}] \quad (\text{subst})
\end{array}$$

Case $\mathsf{let}\ \langle r,x\rangle = \mathsf{openrgn}\ \langle \rho,v\rangle\ \mathsf{in}\ e \Longrightarrow e[^{\rho,\,v}/_{r,\,x}]$**:**

$$\begin{array}{c}
\Psi;\bullet;\Theta;\bullet \vdash \mathsf{let}\ \langle r,x\rangle = \mathsf{openrgn}\ \langle \rho,v\rangle\ \mathsf{in}\ e \\ \hline
\Psi;\bullet,r{:}\mathsf{R}\ \kappa;\Theta,r\mapsto\vec{\rho};\bullet,x{:}\sigma[^r/_{r'}] \vdash e \qquad \Psi;\bullet;\Theta;\bullet \vdash v : \sigma[^\rho/_{r'}] \qquad \dfrac{\Psi;\bullet;\Theta;\bullet \vdash \langle\rho,v\rangle : \exists r'\in\vec{\rho}.\sigma \quad \bullet \vdash^{CiC} \rho_i : \mathsf{R}\ \kappa}{\dfrac{\Theta \vdash \rho \in \vec{\rho} \quad \bullet \vdash^{CiC} \rho_i : \mathsf{R}\ \kappa}{\bullet \vdash^{CiC} \rho : \mathsf{R}\ \kappa}} \\ \hline
\Psi;\bullet;\Theta[^\rho/_r],\rho\mapsto\vec{\rho}[^\rho/_r];\bullet,x{:}\sigma[^\rho/_{r'}] \vdash e[^\rho/_r] \qquad \Psi;\bullet;\Theta;\bullet \vdash v : \sigma[^\rho/_{r'}] \quad (r \text{ free in } \Theta) \\ \hline
\Psi;\bullet;\Theta,\rho\mapsto\vec{\rho};\bullet,x{:}\sigma[^\rho/_{r'}] \vdash e[^\rho/_r] \qquad \Psi;\bullet;\Theta;\bullet \vdash v : \sigma[^\rho/_{r'}] \quad (\text{subst}) \\ \hline
\Psi;\bullet;\Theta,\rho\mapsto\vec{\rho};\bullet \vdash e[^{\rho,\,v}/_{r,\,x}] \quad (\Theta\ \text{equiv}) \\ \hline
\Psi;\bullet;\Theta;\bullet \vdash e[^{\rho,\,v}/_{r,\,x}]
\end{array}$$

The last step implicitly uses the fact that $\Theta \vdash \rho \in \vec{\rho}$ which implies that $\Theta \lhd \Theta, \rho \mapsto \vec{\rho}$. The step that derives $\bullet \vdash^{CiC} \rho : \mathsf{R}\ \kappa$ implicitly uses $\bullet \vdash \Theta$ and is easy to show.

Case $\mathsf{iffirst}\ \nu\ (x, t\ .\ e_1)\ (e_2) \Longrightarrow e_2$**:**

$$\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, 0); \bullet \vdash \mathsf{iffirst}\ \nu\ (x, t\ .\ e_1)\ (e_2)}{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, 0); \bullet \vdash e_2}\ (\text{inv})$$

Case $\mathsf{iffirst}\ \nu\ (x, t\ .\ e_1)\ (e_2) \Longrightarrow e_1[{}^{\tau,\,\nu.0}/_{t,\,x}]$**:**

$$\dfrac{\dfrac{\dfrac{\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash \mathsf{iffirst}\ \nu\ (x, t\ .\ e_1)\ (e_2)}{\Psi; \bullet, t{:}\kappa; \Theta, \nu \mapsto (\varphi, n); \bullet, x{:}t\ \mathsf{at}\ \nu.0 \vdash e_1 \qquad \dfrac{\dfrac{\bullet \vdash^{CiC} \nu : \mathsf{R}\ \kappa}{\bullet \vdash^{CiC} \Psi(\nu.0) : \kappa}\ (\vdash \Psi)}{\bullet \vdash^{CiC} \tau : \kappa}\ (\Psi(\nu.0) = \tau)}\ (\text{inv})}{\Psi; \bullet; \Theta[{}^{\tau}/_{t}], \nu \mapsto (\varphi[{}^{\tau}/_{t}], n); \bullet, x{:}\tau\ \mathsf{at}\ \nu.0 \vdash e_1[{}^{\tau}/_{t}]}\ (\text{subst})}{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet, x{:}\tau\ \mathsf{at}\ \nu.0 \vdash e_1[{}^{\tau}/_{t}]}\ (t \text{ free in } \Theta)}{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash e_1[{}^{\tau,\,\nu.0}/_{t,\,x}]}\ (\text{subst})$$

The last step uses the fact that $\nu.0 : \tau\ \mathsf{at}\ \nu.0$ which follows from the fact that $\Psi(\nu.0) = \tau$.

Case $\mathsf{ifnext}\ \nu.n\ (x, t\ .\ e_1)\ (e_2) \Longrightarrow e_2$**:**

$$\dfrac{\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n{+}1); \bullet \vdash \mathsf{ifnext}\ \nu.n\ (x, t\ .\ e_1)\ (e_2)}{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n{+}1); \bullet \vdash \nu.n : \tau\ \mathsf{at}\ \nu.n \qquad \Psi; \bullet; \Theta, \nu \mapsto (\varphi, n{+}1); \bullet \vdash e_2}\ (\text{inv})}{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n{+}1); \bullet \vdash e_2}$$

Case $\mathsf{ifnext}\ \nu.m\ (x, t\ .\ e_1)\ (e_2) \Longrightarrow e_1[{}^{\tau,\,\nu.m+1}/_{t,\,x}]$**:**

$$\dfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash \mathsf{ifnext}\ \nu.m\ (x, t\ .\ e_1)\ (e_2)}{\dfrac{\dfrac{\bullet \vdash^{CiC} \nu : \mathsf{R}\ \kappa}{\bullet \vdash^{CiC} \Psi(\nu.m{+}1) : \kappa}\ (\vdash \Psi)}{\bullet \vdash^{CiC} \tau : \kappa}\ (\Psi(\nu.m{+}1) = \tau) \qquad \Psi; \bullet, t{:}\kappa; \Theta, \nu \mapsto (\varphi, n); \bullet, x{:}t\ \mathsf{at}\ \nu.m{+}1 \vdash e_1}\ (\text{inv})$$
$$\vdots$$
$$\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash e_1[{}^{\tau,\,\nu.m+1}/_{t,\,x}]$$

The last steps are the same as for the proof of $\mathsf{iffirst}$.

$\square$

Lemma A.2.12 (Progress)

If $\vdash (M; \Theta; \Psi; e)$, *either* $e = \mathsf{halt}\ v$ *or* $(M; \Theta; \Psi; e) \Longrightarrow (M'; \Theta'; \Psi'; e')$.

Proof By induction on the typing derivation of e. As was the case for the proof of type preservation, for all the reduction steps that were already in the core language, the proof is the same so I omit it.

Case $\mathsf{let}\ \langle t, x\rangle = \mathsf{open}\ v\ \mathsf{in}\ e$**:**

$$\cfrac{\cfrac{\Psi; \bullet; \Theta; \bullet \vdash \mathsf{let}\ \langle t, x\rangle = \mathsf{open}\ v\ \mathsf{in}\ e}{\Psi; \bullet; \Theta; \bullet \vdash v : \exists t'.\kappa\sigma}\ \text{(inv)}}{v = \langle \varphi, v'\rangle}\ \text{(inv knowing that } \Gamma = \bullet)$$

So we reduce with: $(M; \Theta; \Psi; \mathsf{let}\ \langle t, x\rangle = \mathsf{open}\ \langle \varphi, v'\rangle\ \mathsf{in}\ e) \Longrightarrow (M; \Theta; \Psi; e[{}^{\varphi,\, v'}\!/_{t,\, x}])$

Case $\mathsf{let}\ \langle r, x\rangle = \mathsf{openrgn}\ v\ \mathsf{in}\ e$**:**

$$\cfrac{\cfrac{\Psi; \bullet; \Theta; \bullet \vdash \mathsf{let}\ \langle r, x\rangle = \mathsf{openrgn}\ v\ \mathsf{in}\ e}{\Psi; \bullet; \Theta; \bullet \vdash v : \exists r' \in \vec{\rho}.\sigma}\ \text{(inv)}}{v = \langle \rho, v'\rangle}\ \text{(inv knowing that } \Gamma = \bullet)$$

So we reduce with: $(M; \Theta; \Psi; \mathsf{let}\ \langle r, x\rangle = \mathsf{openrgn}\ \langle \rho, v\rangle\ \mathsf{in}\ e) \Longrightarrow (M; \Theta; \Psi; e[{}^{\rho,\, v}\!/_{r,\, x}])$

Case $\mathsf{iffirst}\ \rho\ (x, t\ .\ e_1)\ (e_2)$**:**

$$\frac{\Psi; \bullet; \Theta, \rho \mapsto (\varphi, n); \bullet \vdash \mathsf{iffirst}\ \rho\ (x, t\ .\ e_1)\ (e_2)}{\rho = \nu}\ (\Delta \text{ is empty})$$

If $n = 0$ we reduce with: $(M; \Theta; \Psi; \mathsf{iffirst}\ \nu\ (x, t\ .\ e_1)\ (e_2)) \Longrightarrow (M; \Theta; \Psi; e_2)$. Otherwise:

$$\frac{n \neq 0 \quad \nu \in \mathit{Dom}(\Theta, \rho \mapsto (\varphi, n)) \quad \Psi; \Theta, \rho \mapsto (\varphi, n) \vdash M}{\tau = \Psi(\nu.0)}$$

so we can reduce with: $(M; \Theta; \Psi; \mathsf{iffirst}\ \nu\ (x, t\ .\ e_1)\ (e_2)) \Longrightarrow (M; \Theta; \Psi; e_1[{}^{\tau,\, \nu.0}\!/_{t,\, x}])$

Case $\mathsf{ifnext}\ v\ (x, t\ .\ e_1)\ (e_2)$**:**

$$\cfrac{\cfrac{\Delta = \bullet}{\rho = \nu} \quad \cfrac{\Psi; \bullet; \Theta, \rho \mapsto (\varphi, n); \bullet \vdash \mathsf{ifnext}\ v\ (x, t\ .\ e_1)\ (e_2)}{\Psi; \bullet; \Theta, \rho \mapsto (\varphi, n); \bullet \vdash v : \tau\ \mathsf{at}\ \rho.m}\ \text{(inv)}}{\cfrac{\Psi; \bullet; \Theta, \nu \mapsto (\varphi, n); \bullet \vdash v : \tau\ \mathsf{at}\ \nu.m}{v = \nu.m}\ \text{(inv knowing that } \Gamma = \bullet)}$$

If $n = m{+}1$ we reduce with: $(M;\Theta;\Psi;\mathsf{ifnext}\ \nu.m\ (x, t\ .\ e_1)\ (e_2)) \Longrightarrow (M;\Theta;\Psi;e_2)$.

Otherwise:

$$\dfrac{\Psi;\Theta,\nu\mapsto(\varphi,n)\vdash M \qquad \dfrac{\dfrac{\Psi;\Theta,\nu\mapsto(\varphi,n)\vdash M \qquad \dfrac{\Psi;\bullet;\Theta,\nu\mapsto(\varphi,n);\bullet\vdash \nu.m:\tau\ \mathsf{at}\ \nu.m}{\Psi(\nu.m)=\tau}}{m{+}1\leq n} \qquad n\neq m{+}1}{m{+}1<n}}{\tau=\Psi(\nu.m{+}1)}$$

so we can reduce with: $(M;\Theta;\Psi;\mathsf{ifnext}\ \nu.m\ (x, t\ .\ e_1)\ (e_2)) \Longrightarrow (M;\Theta;\Psi;e_1[{}^{\tau,\,\nu.m+1}/_{t,\,x}])$

$\square$