IMPROVING EFFICIENCY AND SAFETY OF PROGRAM GENERATION

BY

TANKUT BARIS AKTEMUR

B.S., Bilkent University, 2003
M.S., University of Illinois at Urbana-Champaign, 2005

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Associate Professor Samuel Kamin, Chair
Associate Professor Vikram Adve
Assistant Professor Darko Marinov
Associate Professor Grigore Roşu
Professor Peter Sestoft (IT University of Copenhagen, Denmark)

Abstract

This dissertation presents three techniques that allow faster generation of programs, and shows that Program Generation (PG) is about writing programs that write programs. A program generator composes various pieces of code to construct a new program. When employed at runtime, PG can produce an efficient version of a program by specializing it according to inputs that become available at runtime. PG has been used in a wide range of applications to improve program efficiency and modularity as well as programmer productivity.

There are two major problems associated with PG: (1) Program generation has its own cost, which may cause a performance loss even though PG is intended for performance gain. This is especially important for runtime program generation. (2) Compilability guarantees about the generated program are poor; the generator may produce a type-incorrect program. In this dissertation we focus on these two problems. We provide three techniques that address the first problem. First, we show that just-in-time generation can successfully reduce the cost of generation by avoiding unnecessary program generation. We do this by means of an experiment in the context of marshalling in Java, where we generate specialized object marshallers based on object types. Just-in-time generation improved the speedup from 1.22 to 3.16. Second, we apply source-level transformations to optimize the execution of program generators. Up to 15% speedup has been achieved in runtime generation time for Jumbo, a PG system for Java. Third, we provide a technique to stage analysis of generated programs to perform a portion of the analysis at compile time rather than completing the entire analysis at runtime. We also give experimental evidence via several examples that this technique reduces runtime generation cost. To address the second problem of PG, we first show that operational semantics of record calculus and program generation are equivalent, and that a record type system can be used to type-check program generators. We also show that this is true in the presence of expressions with side-effects. We then make use of an already-existing record calculus feature, subtyping, to extend the program generation type system with subtyping constraints. As a result, we obtain a very expressive type system to statically guarantee that a generator will produce type-safe code. We state and prove the theorems based on an ML-like language with program generation constructs.

Beauty will save the world.
—Dostoevsky

Acknowledgments

Acknowledging all the people who touch the making of a dissertation requires a long list; mine is no exception.

I am indebted to my advisor Prof. Sam Kamin. He has always been there to help me out and show me the direction to take whenever I felt stuck. He has not only been a mentor and teacher, but also provided my financial support for several semesters, and made it possible for me to attend conferences. I also learned a lot from him about teaching during the time I was appointed as his teaching assistant.

My committee members, Prof. Vikram Adve, Prof. Darko Marinov, Prof. Grigore Roşu, and Prof. Peter Sestoft provided very valuable feedback on my thesis. What I learned in the courses they taught, namely the compiler courses by Prof. Adve, program testing and analysis course by Prof. Marinov, and programming language courses by Prof. Roşu, has been very useful in this dissertation. Prof. Sestoft introduced me to the problem of library specialization that motivated the work presented in Chapter 5.

Prof. Elsa Gunter has been very kind to generously offer her time to help me with the mathematics in my dissertation. I was fortunate enough to also be her teaching assistant for several semesters. She taught me priceless skills in teaching and in dealing with difficult situations with students.

I learned very useful teaching techniques from Professors Lawrence Angrave and Craig Zilles during the time I realized that dealing with the freshmen is not easy at all.

Prof. Chung-chieh Shan pointed out that pluggable declarations in Chapter 5 can be treated as syntactic sugar.

Paul Adamczyk provided comments not only on this dissertation, but also on politics, history, geography and philosophy in general. I thank him for turning the Friday fish into a tradition, for inviting me to Argentina (with special thanks to Federico Balaguer and his family), for being a guide in the Kickapoo state park, for coming up with the idea of going to the Appalachian mountains, for bringing bottles of Żubrówka, and for providing sarcasm as a free service.

I had several discussions with Philip Morton and Michael Katelman — my office mates, collaborators, and fellow students — that shaped an important part of work documented here. I thank Mike for also meeting me for lunch and making me socialize by introducing me to other fellow students.

I would like to thank several people for doing class projects, having lunch with me, being wonderful fellow TAs, having great conversation, etc: Rob Bocchino, Nicholas Chen, Tanya Crenshaw, Danny Dig, Chucky Ellison, Brian Foote, Munawar Hafiz, Mark Hills, Dongyun Jin, William Mansky, Chris Osborn, Jeff Overbey, Andrei Popescu, Maurice Rabb, Traian Serbanuta, Anna Yershova.

I am sincerely thankful to my Turkish friends Emre Akbaş, İnci and Burak Güneralp, Norma Linton, Lale Özkahya, Özgül and Onur Pekcan, Nejan and Süleyman Sarıhan, Sonat Süer, Çiğdem Şengül, Derviş Can Vural, and Serdar Yüksel for being a family to me in Urbana-Champaign.

I appreciate the great atmosphere of the Grainger Engineering Library, Krannert Center for the Performing Arts, Champaign Public Library, Cafe Kopi, Esquire, Legends, and Blind Pig.

I cannot truly express how grateful I am to my family, especially to my parents Melahat and Emin Aktemur. Feeling their support was the most important thing during my years in Illinois. This dissertation is dedicated to them.

Last, but not the least, I thank my beautiful fiancée Sevil Şenol, for her love, support and patience.

A part of the work presented in this dissertation was partially funded by NSF under the grant CCR-0306221.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Program Generation (PG) is about writing programs that write programs. If a program's structure is so routine that it can be built by an algorithm, it is natural to use PG to manufacture the program because this improves program reusability and performance as well as programmer productivity, while decreasing human error [CE00]. PG has been used in (or proposed for) a wide range of applications including implementation of a staged interpreter [Tah03], increasing efficiency in web servers providing dynamic web content [Lea06], literate programming, faster huffman encoding, and generation of proxy classes [Kam04], fine-controlled loop unrolling, finite state machine generation, and encoder/decoder generation [Kam03], convolution matrices and product line architectures [Cla04], object serialization [AJKC05, NR04], domain-specific language development [COST04], and implementation of customizable libraries [AK09, HZS05] among others. These applications show how broadly PG can be used.

In this dissertation we address two important challenges of PG: the efficiency of program generation, and type-safety of the generated program. This chapter first gives a definition of our problem context in Section 1.1, followed by the problem statement in Section 1.2, and our contributions in Section 1.3. The terminology we use throughout the dissertation is introduced in Section 1.4. We then give a brief history of program generation in Section 1.5. We conclude this chapter with the outline of the dissertation.

1.1 Problem Context

Generation of a program in a PG system is done by combining *program fragments*. A fragment is an arbitrarily sized but parseable piece of code, denoted by the quotation syntax $\langle\cdot\rangle$. Composition of fragments is performed by filling in the *hole*s defined in fragments with other fragments. Holes are defined using the antiquotation syntax $`(\cdot)$. All PG systems share this idea of using a quotation/antiquotation syntax to denote and compose the fragments, inspired from quasiquotations in Lisp [Baw99].

Fragments are first-class values that can be passed around and assigned to variables. Holes get filled in by evaluating an antiquoted expression to a fragment, and splicing that fragment in the place of the hole. What fragments fill in which holes and when this

```
// The generic exponentiation method that computes x^n
int power(int n, int x) {
  int c = 1;
  for(int i=0; i<n; i++) {
    c = c * x;
  }
  return c;
}

// The generator that produces the code for computing x^n for a given n
Code genBody(int n) {
  Code c = ⟨ 1 ⟩;
  for(int i=0; i<n; i++) {
    c = ⟨ `(c) * x ⟩;
  }
  return c;
}

// The generator that produces the code of a method to compute x^5
Code genPower5() {
  return ⟨ int power5(int x) {
              return `(genBody(5)); // will be equivalent to 1*x*x*x*x*x
           } ⟩;
}
```

Figure 1.1: Jumbo code that generates a function to take the fifth power of its argument.

happens is determined by the generator, a program that contains quoted fragments. This is the mechanism of combining fragments to obtain the eventual generated program.

A sample program generator is given in Figure 1.1, where we show the classical exponentiation example written in Jumbo [KCJ03], an extension of Java with program generation facilities. In this example, a version of the generic exponentiation method that is specialized for a fixed exponent value of 5 is generated. Note that, because in the process of generating the code, the for-loop is eliminated, this code would execute faster than the generic method. (Note that for large values of the exponent n, we may end up with very large code that may in fact be less efficient due to cache misses.) Fragments in Jumbo have the type Code.

Most program generation systems provide an eval or run construct that compiles and immediately executes a program at runtime without terminating the session in execution. This makes it possible to specialize a program according to inputs that become available only at runtime. For this reason, the major motivation behind using program generation is to obtain better efficiency. The exponentiation example above is such a case. However, as listed at the beginning of this chapter, there exist several other usage areas as well.

1.1.1 Generality of Program Generation

In this dissertation, we are interested in PG systems that provide a high degree of "generality". There are two dimensions of generality.

- *Generality inside quotations:* Can arbitrary fragments be quoted? Does the system allow defining expressions as fragments? How about statements and/or declarations? Can a fragment have free variables? Can there be quoted code values inside quoted code values? Can you fill in a hole inside a loop with a break statement?

- *Generality outside quotations:* Can the fragments be passed around as first class values? Is there a construct that allows building fragments using loops?

In principle, we are interested in PG systems that give the answer "yes" to all the questions above — systems that allow the users to "generate anything, in any way they want." This is not always possible. Strong type-safety requirements almost always put limitations on what can be generated. In this dissertation, we use a program generation system that, unless otherwise noted, does not compromise the following properties, giving it a high-degree of generality.

- Free variables are allowed inside quotations. These variables may get captured when the fragment fills in a hole.

- Quotations can be *multi-leveled*; having quoted codes inside a quoted code is allowed. This makes it possible to generate generators.

- Both expressions and statements (including declarations) can be quoted.

- The meta-language that is used outside the quotations to manipulate code pieces is a Turing-complete programming language.

- Fragments are first-class citizens; they can be passed around, returned from functions, assigned into variables.

Not every PG system provides generality at this degree, either because of implementation issues or more fundamental reasons such as generating more efficient code or providing stronger formal properties. Where appropriate, we comment on other systems' restrictions on generality in the upcoming chapters when evaluating the related work.

In this dissertation, we use Jumbo [KCJ03] as the PG system. Because Jumbo is a complex system providing all the features of a real-world language –Java– we assume a simplified version of it whenever this simplification does not cause a loss of generality. In Chapter 5, we use an ML-like functional language that has program generation facilities giving the generality properties listed above.

1.1.2 PG by Partial Evaluation vs. PG by Program Construction

Program generation can be classified into two categories: *PG by partial evaluation* [ACK03, CX03, Dav96, DP96, KKcS08, MTBS99, TN03, YI06] and *PG by program construction* [Baw99, HZS05, HZS07b, KCJ03, KYC06, OMY01, PHEK99, Rhi05, ZHS04]. These two approaches to program generation require different mindsets when programming.

PG by partial evaluation is based on the ideas of partial evaluation. This PG style is about *delaying* the execution of some part of a code while regularly evaluating the other parts. The programmer may explicitly annotate the program to indicate which part to delay or not to delay, as opposed to partial evaluation's implicit binding-time analysis [JGS93, NN92]. This kind of PG enjoys the "erasure property" [DP96]: a valid program can be obtained if all the annotations are erased. This means that there can be no unbound variable in a program, even in the delayed fragments (i.e. inside quotations).

PG by program construction is about *building* new programs by composing program fragments. Programmers again explicitly define fragments and how they are combined. There is no erasure property; removing annotations may leave a meaningless, or even unparsable program. Jumbo falls into this category.

An important difference between PG by partial evaluation and PG by program construction is *variable hygiene*. In PG by program construction, free variables in a fragment may get "captured" and bound when the fragment is spliced into a context. Composition of code values is intentionally *unhygienic*. This is a property common to all PG by program construction systems. In PG by partial evaluation, however, variable capturing is forbidden. The binding of a variable is statically known, and variables are alpha-converted to avoid capturing; composition of code values is intentionally *hygienic*. For example, the program $\mathsf{let}\ f = \lambda c.\langle \lambda x.x + \,\grave{}\,(c)\rangle\ \mathsf{in}\ \langle \lambda x.\,\grave{}\,(f\langle x\rangle)\rangle$, written in ML-like syntax, yields a value that is alpha-equivalent to $\langle \lambda y.\lambda z.z + y\rangle$ if evaluated in MetaOCaml [TCLP] — a PG-by-partial-evaluation system. On the other hand, the output is $\langle \lambda x.\lambda x.x + x\rangle$, if evaluated in λ^{open}_{poly} [KYC06] — a PG-by-program-construction system.

1.2 Problem Statement

Program generation systems typically facilitate generation and immediate use of code at runtime. This allows for taking runtime inputs into consideration to generate a more efficient version of a program. However, runtime generation has its own cost. Runtime generation should be employed only if the efficiency gain from the specialized program exceeds the generation cost. Consider the exponentiation example given in Figure 1.1. If we want to take the fifth power of a single number, it would be pointless to generate a function specialized for that purpose; the generation cost would be much bigger than the cost of the computation using a generic exponentiation function. However, if we will take

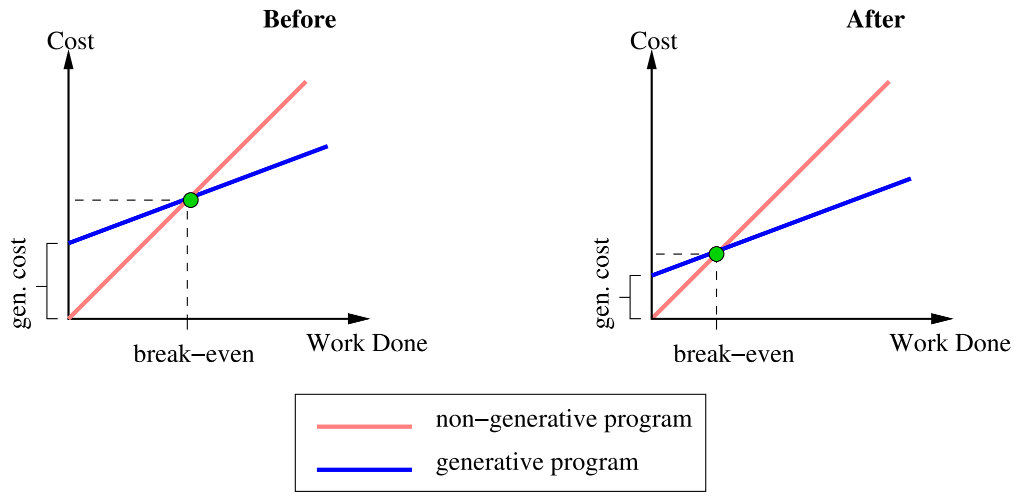

Figure 1.2: One of our goals is to decrease the code generation cost so that the break-even point will be lower.

the fifth power of thousands of numbers, we may want to generate the specialized function, because it runs faster than the generic power function and at some point it is going to compensate for its generation cost. The moment when the generated program starts to pay off is called the *break-even point* (or *crossover point*). The cost of runtime generation needs to be decreased as much as possible so that the break-even point will be lower (see Figure 1.2). A low break-even point increases the confidence that PG will be profitable rather than costly. This is the first challenge of program generation.

Problem 1: *Program generation has a cost. Because of this cost, although intended to speed up programs, PG may in fact cause slow-down in some circumstances.*

In PG, it is not easy to reason about the properties of the generated code by looking at the generator, because in the generator we only have partial information. On the other hand, users would like to have some assurance of safety properties about the product. If the generator produces a program that does not even compile, it would frustrate the user, and embarrass the programmer of the generator. Therefore it is desirable to guarantee that a generator will produce type-safe code. This is the second challenge of program generation.

Problem 2: *PG does not provide guarantees about the product. A generator may produce ill-formed code that does not even compile.*

Note that the notion of correctness can be expanded beyond type-safety to include behavioral correctness. That is, how do we make sure that the generated code runs as expected? However, due to its meta-nature, this problem seems tough; the state of the art in program generation is still investigating ways for obtaining better type correctness properties. Therefore, we focus on type checking in this dissertation, and put more advanced behavioral correctness properties out of our scope.

1.3 Contributions

In this dissertation we study the two problems outlined in the previous section. We develop techniques to

- make PG more profitable by reducing the cost of generation, and
- make PG safer by guaranteeing type-safety of the generated code.

In particular, our contributions are summarized as follows:

- We provide empirical results showing that just-in-time program generation can be effective in avoiding unnecessary runtime generation cost, and thus lowering the break-even point. For our experiment we study marshalling in Java, where we generate marshallers specialized to certain object types. We show that just-in-time program generation can improve the speedup obtained by PG from 1.22 to 3.16.
- We show that source-level rewriting techniques can be applied to partially evaluate generators at compile-time for faster generation at runtime. We obtain up to 15% speedup in runtime generation time in Jumbo. In a prototype system, we had achieved up to 60% speedup. We provide a discussion of what prevents us from getting higher speedup in Jumbo.
- We develop a technique to stage static analyses to partially analyze generated code at compile-time and thus reduce runtime generation cost. We provide analysis frameworks for both forward and backward analysis, and define several instantiations of these frameworks to concrete analyses. Depending on the characteristics of the application, we obtain various speedup results that are as high as an order of magnitude.
- To address the safety problem of PG, we first define a translation that converts program generators to programs in the record calculus. We then prove a theorem stating that the operational semantics of the record calculus is equivalent to the operational semantics of program generation. This leads to the fact that a record calculus type system can serve as a sound program generation type system. We prove that such a type system is equal to an existing PG type system, λ_{poly}^{open}[KYC06]. We show that the results hold even in the presence of side-effectful expressions. By using existing knowledge about subtyping in the record calculus, we are able to have subtyping for program generation as well. All these properties yield a very powerful PG type system.

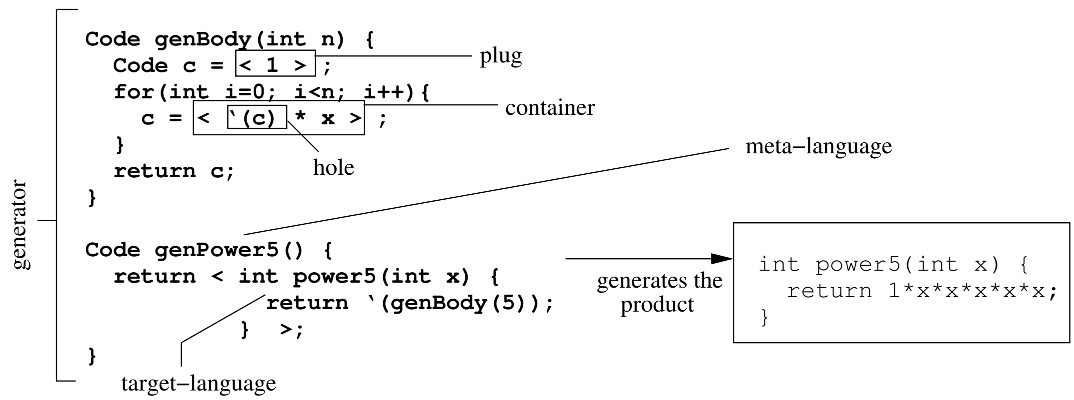

Figure 1.3: PG terms shown on the exponentiation example.

Program generation has a third challenge, namely the size of the generated code (or the generator-size problem in partial evaluation [JGS93, GJ95]). Because specialization usually includes inlining of functions or unrolling of loops, the size of the specialized code may get undesirably large, which may negatively affect the behavior of the hardware instruction cache and register allocation. This problem of program generation is out of the scope of our thesis.

1.4 Terminology and Notation

In this section we introduce the terminology and notation we use in this dissertation.

We use the quotation syntax $\langle\cdot\rangle$ to denote a *program fragment*, and the antiquotation syntax $`(\cdot)$ to denote a *hole* inside a fragment. The synonyms for program fragment are *quoted fragment*, *quoted code*, *code value*, *program piece*, or *code piece*. Quotations/antiquotations in a program are also called *annotations*. A fragment that fills in a hole is called a *plug*. A fragment containing holes to be filled in with plugs is called a *container* fragment. Note that once its holes are filled, a container fragment may itself become a plug. The language used inside the fragments is called the *target* or *object language*. The language that is used for managing the composition of fragments is the *meta* or *host language*. For example in Jumbo [KCJ03], the target and the meta language are the same: Java. In MetaAspectJ [ZHS04], the meta language is Java whereas the target language is AspectJ.

A quoted fragment is said to be in the next *stage* or *level*. The first stage (i.e. the stage of the meta-language) is stage 0. A program that contains quoted fragments is called a *generator* or a *(multi-)staged program*, and the program that will be generated is called the *product* or the *generated program*.

Some of the terms are illustrated in Figure 1.3.

In this dissertation we often write Jumbo code. However, to reduce notational clutter, we use a simpler syntax than Jumbo's. In particular, we have the following differences:

- We use the quotation syntax $\langle ... \rangle$ instead of Jumbo's `$<...>$`.

- Jumbo uses labeled antiquotations to denote the syntactic category a hole represents, such as `Expr(...) and `Stmt(...). Jumbo requires this information for its parser to work[1]. We simply use the plain antiquotation $`(...)$.

- Jumbo provides antiquotation syntax to "lift" primitive values and strings to the next level: `Int(...), `Char(...), etc. These are similar to the cross-stage-persistence operator (%) in MetaML [TS00, TN03], and analogous to the $\mathsf{lift}(...)$ operator in λ^{open}_{poly}[KYC06], which raises a value to the next level by quoting it. We use $\mathsf{lift}(...)$. In this notation, Jumbo's `Int(...) becomes equivalent to $`(\mathsf{lift}(...))$.

Based on when the product is run, there are two kinds of program generation: *static* (or compile-time) and *dynamic* (or runtime). In static generation, the generator yields the product, and then the product is executed in a totally new runtime environment as a separate process. It may be shipped to a remote client to be executed there. In dynamic generation the product is directly brought into the executing process of the generator. The generator can then directly refer to the result of executing the product; this way, runtime inputs of the program can be taken into consideration for optimization. An illustration of the two kinds of program generation is given in Figure 1.4.

Program generation systems that facilitate dynamic generation typically provide a construct or an operator to run a code value at runtime. We will use $\mathsf{run}(\cdot)$, unless otherwise noted.

There are several terms used for "program generation". We will interchangeably use these terms to avoid repetition. These terms include *multi-stage programming*, *multi-level programming*, *meta-programming*, *code generation*, and *runtime program generation*[2] if dynamic generation is to be emphasized.

An important distinction is made in program generation between fragments that contain free variables and fragments that do not. The former is referred as *open code*; the latter is *closed code*. There is, an important difference in the notion of "open code" in the context of PG by partial evaluation and PG by program construction. Even though many PG by partial evaluation systems allow fragments to have free variables, these variables have to be bound by an outer binding — otherwise the "erasure property" cannot be satisfied. Systems following the PG by program construction philosophy do not have this requirement;

[1]MetaAspectJ [ZHS04] uses a more complicated parser with inference to automatically infer these categories when possible.

[2]A common term for this is "runtime code generation" (RTCG). We prefer to use "runtime program generation" (RTPG) instead to emphasize that the generation process is controlled by the programmer by defining pieces of source program, as opposed to the more low-level and machine-directed feeling implied by the term "code generation" [Kam04, footnote 2].

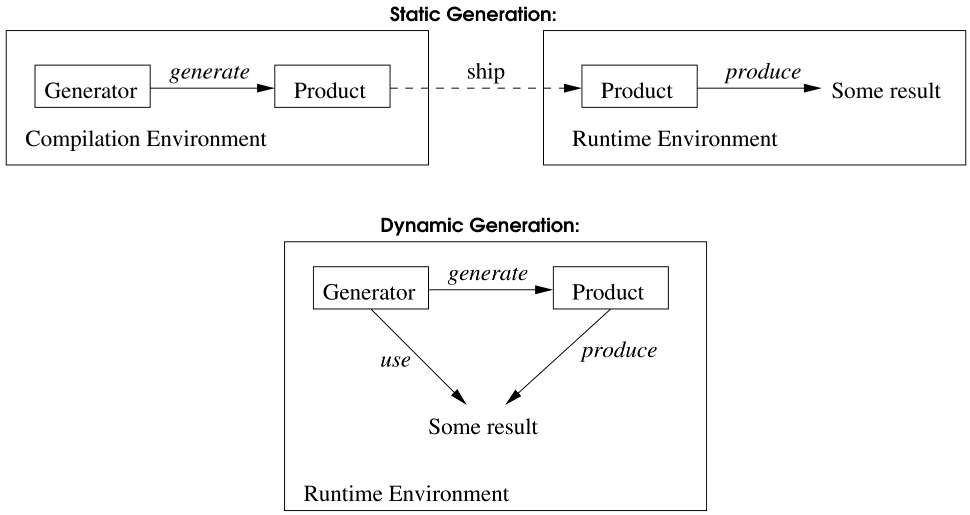

Figure 1.4: High-level illustration of static and dynamic generation.

Example program	Classification	Acceptable in PG by partial evaluation	Acceptable in PG by program construction
$\langle \lambda x.x + 1\rangle$	*closed*	Yes	Yes
$\langle \lambda x.`(\langle x + 1\rangle)\rangle$	*open*	Yes	Yes
$(\lambda c.\langle \lambda x.`(c)\rangle)\langle x + 1\rangle$	*freely-open*	No	Yes

Table 1.1: Distinguishing between closed, open, and freely-open code.

a fragment that contains free variables that are not in the scope of any binding is acceptable to these PG systems. Free variables get bound when the fragment is plugged inside a fragment that provides the bindings for those variables. To distinguish between these two interpretations of "open", we use the term *open code* to refer to the meaning in the context of PG by partial evaluation: free variables inside fragments are allowed but they have to be inside the scope of an outer binding; and the term *freely-open code* to refer to the meaning in the context of PG by program construction: free variables inside fragments are allowed even if they are not in the scope of a binding. A summary of this distinction with examples is given in Table 1.1.

1.5 Brief History of Program Generation

PG by partial evaluation

The roots of "PG by partial evaluation" go back to Futamura's work from 1971 where he proposes partially evaluating an interpreter to obtain a compiler [Fut99]. Even though the idea has been known, it took almost fifteen years to put it into practice [JSS85]. Early languages that were used for partial evaluation had two levels. Theoretical aspects of these languages, including their denotational semantics and analysis using abstract interpretation are studied in detail by Nielson and Nielson [NN92]. Glück and Jørgensen generalized the two-level approach to multi-levels and showed that significant efficiency gains can be obtained [GJ95, GJ97]. Their technique addresses the problems of partial evaluation related to code-size expansion and the cost of generation. Besides practical aspects of partial evaluation, checking type-safety of partial evaluators has been a problem long-studied by researchers. Davies and Pfenning addressed this problem and generalized Nielson and Nielson's two-level typing to multi-levels. They define two languages, $\lambda^{\Box}$ [DP96] and $\lambda^{\bigcirc}$ [Dav96], where binding times (i.e. when an expression should be evaluated) are explicitly specified by means of annotations as opposed to the previous papers' implicit annotations. $\lambda^{\Box}$ only allows declaration of *closed code* via annotations, and provides a $\mathsf{run}()$ operator — which did not exist in Nielson and Nielson's or Glück and Jørgensen's languages — to evaluate code fragments. $\lambda^{\bigcirc}$ allows definition of *open code* —code with free variables— but does not include the construct of $\mathsf{run}()$ in order to preserve soundness.

Working with closed code only is not practical and $\mathsf{run}()$ is a very valuable feature. Hence, there has been a significant amount of work in the literature to combine $\mathsf{run}()$ with open code. Taha and Sheard's MetaML [TS00, TS97] is such a language with a sophisticated type system that refines the notion of open code. However, MetaML has the *scope extrusion* problem, where the type system fails to detect evaluation of open code fragments. A typical example is $\langle \lambda x.\,`(\mathsf{run}\langle x \rangle)\rangle$. An Idealized ML (AIM) [MTBS99] addresses this problem by splitting the types for codes into two —open and closed— and only allows running code with closed type. This approach is further extended in $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$ [CMT00] to handle references as well. To avoid scope extrusion, both AIM and $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$ make conservative assumptions and may reject code that would otherwise be safe to execute[3].

Nanevski [Nan02], as opposed to MetaML's approach of refining the notion of open code, relaxed the definition of closed code to allow free variable inside quotations while still providing $\mathsf{run}()$. In his system, free variables of a fragment are represented with a new semantic category, *names*. His type system allows fragments with free variables to be used only for filling in holes — they cannot be run.

Taha and Nielsen defined *environment classifiers* to overcome the restrictions of AIM about having closed code [TN03]. In their language, called λ^{α}, code types are annotated with labels that specify the environment they belong to. These decorations keep track of free variables that occur in code fragments. Calcagno, Moggi and Taha further developed a type inference algorithm for a slightly limited but still expressive version of λ^{α} [CMT04] to eliminate the need to enter type annotations manually.

Yuse and Igarashi developed $\lambda^{\bigcirc\Box}$ [YI06] as a language that combines features from $\lambda^{\Box}$ and $\lambda^{\bigcirc}$. Using this language they showed the close relation of program generation

[3]See [CX03, §6] for an example.

constructs and linear-temporal logic.

Kameyama, Kiselyov and Shan proposed a transformation from the two-stage version version of λ^{α} to System F with tuples [KKcS08]. This allows for type-checking a staged expression using System F. A similar idea was employed by Chen and Xi [CX03], where they translate quoted code values to first-order abstract syntax.

Attardi, Cisternino and Kennedy [ACK03] took a detour from the languages of the formal world, and added partial evaluation facilities to C#.

PG by program construction

"PG by program construction" has had more diversity than PG by partial evaluation in the styles and contexts of the PG systems designed. The oldest program construction facility is the quasiquotations in LISP. According to Bawden [Baw99], the history of quasiquotations goes as far back as 1940s, and integration and popular use of program generation via a quotation/antiquotation mechanism in LISP started in the 1970s.

Jumbo [KCJ03, Cla04] is an extension of Java with program generation facilities. It can compile Java 1.4. Jumbo provides a very high degree of generality. Almost any parseable fragment can be quoted/antiquoted.

MetaAspectJ [ZHS04] is another system where the meta-language is Java. It outputs AspectJ [Asp] source code. MetaAspectJ is similar to Jumbo in the sense that it supports high generality. It has a sophisticated parser that can infer syntactic categories when possible to eliminate the need to enter these categories manually.

The type-checking problem in the context of "PG by program construction" has gained attention more recently than PG by partial evaluation. DynJava [OMY01] is a program generation system for Java, where code values have to be explicitly decorated with type information. Cyclone [SGM^{+}03] employs a template-based approach, where programs are constructed by combining pre-compiled fragments. Code fragments are not first-class citizens; all the code generation occurs local to a generator function. This reduces flexibility and generality, but also helps to make better approximations about the generated program by taking the control flow of the generator into account. Tempo [CLM04] and `C [EHK96, PHEK99] are other examples to template-based program generation.

Rhiger defined a sound type system where a code value is given a type that contains an environment to carry the types of the free variables occurring in the code [Rhi05]. Kim, Yi and Calcagno improved this type system by extending it with references, let-polymorphism, and hygienic variables [KYC06]. They also gave a principal type inference algorithm. In Chapter 5, we use their type system as a starting point.

There are other program generation systems that are motivated by various needs and ideas. Huang, Zook and Smaragdakis's SafeGen [HZS05] has a type system that is designed to detect whether a variable declared in a code fragment is available for use in another fragment. With this motivation, it allows the programmer to define first-order logic formulas using built-in predicates and functions. A logical property based on these formulas is then fed into a theorem prover. If the property is satisfiable, SafeGen concludes that the generated program will be type safe. In SafeGen, code fragments are not first-class citizens. Developed by the same authors, MJ [HZS07b] and cJ [HZS07a] are two languages where code fragments are included or excluded from the produced code according to predicates specified based on reflective properties. Roughly, MJ provides a "for" construct to iterate over the members of a class, and cJ provides an "if" construct to emit code conditionally. They are designed for Java-like languages. Code fragments are class-members — either fields or methods. CTR [FCL06] is another system that uses reflection to collect properties of existing classes, and uses these properties to generate code.

Kästner, Apel and Kuhlemann [KAK08] approached program generation from a different perspective. Instead of composing fragments, they *take out* fragments from existing code. They developed a system, CIDE, in which fragments can be marked with color codes. Color annotations are used as a mask to retain or exclude code fragments.

1.6 Outline of the Dissertation

Chapters 2 through 4 discuss techniques addressing the first problem of PG: the cost of generation. In Chapter 2, we present an empirical study showing that the idea of just-in-time generation effectively avoids unnecessary program generation. We do this experiment in the context of marshalling. In Chapter 3 we apply source-level transformations on program generators to reduce runtime generation cost. In Chapter 4 we present a framework that naturally leads to staging of dataflow analyses. We show how this technique reduces the runtime cost by performing a portion of the generated program's analysis at compile time.

We focus on the second problem of program generation, namely the type-safety of the generated program, in Chapter 5. This chapter shows that the problem can be effectively addressed using record calculus.

We give our conclusions and discuss future research ideas in Chapter 6. The proofs of key lemmas and theorems are given in the Appendix.

Chapter 2

Just-in-time Program Generation

Runtime program generation (RTPG) allows for producing at runtime an efficient version of a program that is specialized according to runtime parameters. It is typically the case that the generated program has to be executed often to compensate its generation cost. If this cannot be satisfied, the break-even point cannot be reached, which means that RTPG actually slows down the application instead of speeding it up. This is the first problem of program generation we listed in Chapter 1. There are two main approaches one can take to address this problem:

- The specialized program should not be generated if it cannot pay-off its generation cost.
- The cost of runtime generation should be reduced so that a specialized program reaches its break-even point after fewer executions.

In this chapter we empirically investigate the first approach. In particular, we apply the speculative idea of just-in-time compilation: if a case has been seen several times, it is likely that the same case will appear several more times in the future. Using this heuristic, we avoid specialization for a particular case unless it is likely that we will face the same case many times in the future. We perform our experiment in the context of marshalling.

Marshalling is the term used for saving the internal data of an application in an external form. Once marshalled, objects can be passed to other applications. Java RMI (remote method invocation) and CORBA are examples of systems which marshal data for transmission to remote machines. Another term for marshalling is *serialization*. The reverse process is called *unmarshalling*. Serialization generally involves writing large amounts of data, and so is often a performance bottleneck. (According to [NPH99], Java serialization accounts for 25–65% of a remote method invocation.) It can be heavily optimized for any particular type of data. However, optimizing a general-purpose marshaller is difficult because the format of the data to be marshalled is not known at compile-time. Such marshallers are guided by a description of the data that becomes available only at run-time; it is provided either by the client of the marshalling code, or, as in Java, by the language's reflection mechanism. These reasons make marshalling a natural fit as an RTPG application.

In this chapter, we apply RTPG to the problem of optimizing marshalling in Java [MvNV+01, VP03, vNMW+05] using Jumbo [KCJ03, Cla04, Kam03]. We base our implementation on the serializer found in Kaffe [Kaf]. We first investigate marshalling both homogeneous and heterogenous data using specialized serializers but without any just-in-time generation. We then experiment with just-in-time generation using different threshold values. Our empirical findings in this chapter give the following results:

- RTPG is an effective way to achieve significant speedup in marshalling.
- Just-in-time program generation successfully reduces the cost of runtime generation by avoiding unneeded generation.
- The overhead of using just-in-time generation is negligible.

This chapter is organized as follows: In Section 2.1, we discuss marshalling in Java in more detail and give some ideas about where RTPG might help. Section 2.2 shows how Jumbo can be used to implement the suggestions made in Section 2.1. Section 2.3 gives performance comparisons between serialization with and without RTPG. We marshal large, homogeneous and near-homogeneous collections, and heterogeneous collections. In Section 2.4, we discuss usage of just-in-time program generation to reduce the cost of run-time compilation for heterogeneous data as well as its effects on homogeneous data serialization. In Section 2.5, we discuss how our technique applies to Sun's implementation of serialization. Finally, Section 2.6 reviews related work and Section 2.7 concludes this chapter.

The work presented in this chapter has been published in GPCE '05 [AJKC05].

2.1 Marshalling in Java

Java provides a simple API for serialization. A Java programmer doesn't need to write any serialization code, but must simply declare her classes to implement the marker interface `java.io.Serializable`. If a class implements this interface, an instance can be marshalled by passing it to `java.io.ObjectOutputStream`'s (OOS) `writeObject()` method.

Sun provides a specification of serialization [Javc], and an implementation. However, that implementation uses native methods, written in C/C++, to gain efficiency. Therefore, it is not appropriate for our experiment. An implementation in pure Java[1] is provided by Kaffe[Kaf]; we start our study there.

Throughout this chapter we refer to Sun's and Kaffe's implementation as Sun OOS and Kaffe OOS, respectively. The implementation for marshalling which uses RTPG is referred

[1] Actually there is one call to a native method, to test whether a class has a static initializer. This test is not available in the reflection API [Kaf].

```
writeObject(obj) { // method in Kaffe OOS
  if obj = null {
    writeNull;
  } else if obj was already written { // look up the object in the hashtable
    write object handle
  } else if obj is an instance of Class or String {
    write obj according to the specification for that particular case
  } else if obj is an Array {
    foreach element i in obj
      writeObject(i); // a recursive call
  } else if obj is an instance of ClassDescriptor {
    writeClassDescriptor(obj);
  } else {
    // recursive call to serialize class descriptor
    writeObject(classDescriptor of obj);

    // then write contents of the object
    if obj is Serializable {
      foreach classDescriptor in the class hierarchy of obj
        foreach field in classDescriptor
          if field is primitive
            writePrimitive(field);
          else
            writeObject(field); // recursive call
    } else { throw Exception("obj is not serializable"); }
  }
}
```

Figure 2.1: The pseudo code that outlines Kaffe OOS's writeObject() method.

as Jumbo OOS. (In fact there are two versions of Jumbo OOS, one based on Kaffe's implementation and the other based on Sun's implementation, but it will be clear from the context to which one we are referring.) When it doesn't matter which OOS we are referring to, we just say OOS.

We now explain Java serialization in detail, to highlight the places that can be optimized by RTPG. The serialization format is roughly as follows: For each object, first write a descriptor for its class and then write the object's fields; primitive fields are written directly, and object fields are written recursively using the same format. To prevent outputting multiple copies of class descriptors or objects – and to avoid infinite loops – each class and object is assigned an id number, or *handle*; every class and object written is stored in a hashtable the first time it is seen, and only its handle is output on subsequent sightings. The pseudo code in Figure 2.1 outlines Kaffe OOS's writeObject() method.

To summarize, each object is passed through a set of checks: Is the object null? Was it already written to the stream? Is it an array? Was its class descriptor already written? Is it Serializable? Finally, for each class descriptor in the inheritance hierarchy of the object, we find the fields of that class. For each field, if it is primitive, we write the actual value

```
1  writeObject(obj) { // method in Jumbo OOS
2    if obj = null {
3      writeNull;
4    } else if obj was already written { // look up the object in the hashtable
5      write object handle
6    } else {
7      // look for specialized marshaller in the hashtable
8      marshaller = getMarshallerFor(class of obj);
9      if marshaller != null { // marshaller is found
10       marshaller.write(obj);
11     } else if obj is an instance of Class or String {
12       // ... as in Figure 2.1
13
14       if obj is Serializable {
15         // generate specialized marshaller and put it into hashtable
16         marshaller = ProgGen.generateMarshallerFor(obj);
17         storeMarshaller(marshaller);
18
19         // ... as in Figure 2.1
20       }
21     }
22   }
23 }
```

Figure 2.2: The pseudo code that outlines Jumbo OOS's writeObject() method which uses code generation to produce specialized marshallers.

in object directly to the stream. Otherwise, we marshal it by making a recursive call. Note the use of reflection in the above, using class descriptors to discover the fields of the class.

We can optimize the serialization of objects of any class by generating a marshaller specific to it when we first see an instance of that class. After the specialized marshaller is generated, it can be used to serialize subsequent instances. With this alteration, the general marshalling procedure becomes as in Figure 2.2. The difference of this method from that given in Figure 2.1 is that it contains code for generating a specialized marshaller (lines 15-17), and also looking for and using generated marshallers (lines 7-10). As a technical point, the reader will note that a specialized marshaller is not used for marshalling right after it is generated. This is because the first time an object of some type is serialized, the class descriptor of the object has to be fully written. In the subsequent marshallings of objects of the same type, only the handle of the class descriptor is written. The generated code writes only the handle of the class; by not using it the first time an object of some type is written, we delegate the task of fully serializing the class descriptor to the generic marshaller. This frees the generated marshaller from the burden of checking if the class descriptor has been marshalled before.

2.2 Jumbo Code for Marshalling

In section 2.1, we showed how to make use of program generation in Jumbo OOS. In this section we discuss how to write the specialized marshaller generator using Jumbo. We have implemented a class, called ProgGen, which produces the marshallers. Before we explain ProgGen, let's look at the specialized marshaller that would be produced for the following class, representing a linked-list node:

```
public class Node implements Serializable{
  int data;
  Node next;
}
```

Its generated marshaller would be:

```
public class NodeMarshaller implements Marshaller {
  JumboObjectOutputStream oos;
  Field[][] fields;
  int handle;

  public void init(JumboObjectOutputStream oos,
                   Class clazz, int handle) {
    this.oos = oos;
    this.handle = handle;
    ... // initialize fields[][] here - omitted
  }

  public void write(DataOutput stream, Object obj) {
    // Write the OBJECT tag and class handle to the stream
    // These magic numbers are defined in Sun's specification
    stream.writeByte(115);
    stream.writeByte(113);
    stream.writeInt(handle);
    // write the data field
    stream.writeInt(fields[0][0].getInt(obj));
    // send the next field to Jumbo OOS to have it serialized
    oos.writeObject(fields[0][1].get(obj));
  }
}
```

In the code above, fields[ ][ ] holds the field specifiers. The first index corresponds to the position of the class descriptor in the hierarchy, and the second index corresponds to the position of the field in that class descriptor.

Note that Jumbo generates byte code – *not* source code. We have given source code for readability: the byte code generated is just what would be produced by a Java compiler if presented with this source code.

When compared with the original OOS, the specialized marshaller is much simpler. The next field of Node will also be serialized via the specialized marshaller (provided that its run-time type is Node). The marshalling process will end when next is a null pointer or an already serialized object.

ProgGen is obtained by a fairly straightforward massaging of the Kaffe OOS. Basically, ProgGen and Kaffe OOS have code in one-to-one correspondence. However, ProgGen does not write data into a stream like Kaffe OOS does. Instead, it builds the program fragment which does that job. To illustrate, let's examine the writeFields() method of Kaffe OOS. This is the method that actually writes the fields of an object.

```
private void writeFields(Object obj, ObjectStreamClass osc){
  ObjectStreamField[] fields = osc.fields;
  String fieldName;
  Class type;
  for (int i = 0; i < fields.length; i++){
    fieldName = fields[i].getName();
    type = fields[i].getType();
    if (type == Boolean.TYPE)
      realOutput.writeBoolean(
          getBooleanField(obj, osc.forClass(), fieldName));
    else if ... // check for other primitive types
    else        // non-primitive
      writeObject(getObjectField(obj, osc.forClass(),
                  fieldName, fields[i].getTypeString()));
  }
}
```

This method first gets all the fields in the class descriptor osc. Then, by using each field's descriptor, it fetches the value of the field from the object. This is done in getXField() of OOS, where "X" is one of Boolean, Byte, Char, Double, Float, Int, Long, Short, and Object. We show getIntField() below. The methods for other field types are very similar.

```
private int getIntField (Object obj, Class klass, String fname) {
  Field f = getField(klass, fname);
  return f.getInt(obj);
}
```

The getField() method that is used above is implemented as follows (exception-handling is omitted for clarity).

```
Field getField (Class klass, String name) {
  final Field f = klass.getDeclaredField(name);
  AccessController.doPrivileged(new PrivilegedAction() {
    public Object run() {
      f.setAccessible(true);
      return null;
    }});
  return f;
}
```

This work is done for each object field, even if another object of that class was already written. We should not have to find the field specifiers and field types each time, or set the accessibility of the fields to true again and again. Instead we can find the field descriptors once, set their accessibility, and generate code with these descriptors built-in:

```
private Code writeFields(ObjectStreamClass desc, int hier) {
  ObjectStreamField[] fieldDecls = desc.fields;
  Code c = ⟨ ; ⟩;
  for (int i = 0; i < fieldDecls.length; i++){
    Class type = fieldDecls[i].getType();
    if (type == Boolean.TYPE)
      c = ⟨ `(c)
              stream.writeBoolean(
              fields[`lift(hier)][`lift(i)].getBoolean(obj));
          ⟩;
    else if ... // other primitive types
    else // non-primitive type. write the field via Jumbo OOS
      c = ⟨ `(c)
              oos.writeObject(
              fields[`lift(hier)][`lift(i)].get(obj));
          ⟩;
  }
  return c;
}
```

Note that the method above requires hier as an argument. It doesn't need the Object obj parameter anymore, in contrast to the implementation of writeFields in Kaffe OOS. The code shows that if the field is non-primitive, it is passed to the Jumbo OOS to be written. In fact, we keep a one-element cache in the specialized marshaller associated with each non-primitive field; if the run-time type of the field is the same as the one in cache, we call the associated specialized marshaller without passing the object to Jumbo OOS. This saves us from the hashtable lookup that would occur in Jumbo OOS. If there is a cache miss, we pass the object to Jumbo OOS, it performs a hashtable lookup, writes the object and then we update the cache. We do not give this code for the sake of brevity. Our tests showed that keeping this cache in practice brings neither a noticeable overhead nor a speedup. Nevertheless, we opt to keep it in the code. We omit related benchmarking numbers.

After we have the methods that return code pieces to serialize an object, we need to generate the init method[2], which will set up the data in the generated marshaller. This method is where the class handle is assigned to a data member of the serializer and where the fields[ ][ ] matrix is set. Note that this happens only once per generated serializer. This initializer method is constructed using code pieces from Kaffe OOS. Therefore writing this method is again straightforward.

The generated marshallers implement an interface called Marshaller, which defines the methods init and write. Interfaces, or abstract classes, are normally required in Java when ordinary code is to call generated code [Cla04, Kam03, KCJ03].

2.3 Performance

When using RTPG, the cost of run-time program generation must be taken into account. For this cost to pay off, we need to use the generated program a lot; that is, we need to marshal a large data set. Still, the running time of the generated code — excluding compile time — is a useful quantity to know, because it gives the upper limit of speed-up (to which the actual speed-up will converge, if the generated program is used over and over, as the cost of generation will become less and less significant). In this section, we give the performance of specialized marshallers, both including and excluding the cost of run-time compilation.

The performance of marshalling code is highly dependent upon the properties of the data being marshalled. Furthermore, it is not clear what should count as a "realistic" workload for marshalling. Large data sets — which are the ones we most care about, since these will be the most time-consuming to marshal — are likely to consist of large numbers of a few kinds of objects; this would be characteristic of video or audio streams, for example. On the other hand, most data in Java consists of objects of many different types. From the point of view of run-time program generation, these two scenarios have very different performance characteristics. Accordingly, we show benchmarks of both kinds. Specifically, we start by marshalling large, homogeneous collections of a class called Dummy, which has several fields. Then we test a linked-list class, and a class similar to Dummy, but with fields which can contain either of two types of objects (one a subclass of the other). After showing benchmarks for these homogeneous and near-homogeneous collections, we discuss a non-homogeneous data set, containing objects of 66 different classes.

These benchmarks are run as follows: All the tests are executed on a Linux Debian,

[2] Java doesn't provide the ability to pass arguments to the constructors of dynamically loaded classes, so the class can only have a zero-argument constructor [Cla04, Java]. Thus we define a normal method, init, and call it right after the object is created via the zero-argument constructor.

Object Count	Bytes written	Jumbo OOS	Jumbo + compilation	Kaffe OOS	$\frac{\text{Kaffe}}{\text{Jumbo}}$	$\frac{\text{Kaffe}}{\text{Jumbo+comp.}}$
1000	30000	6.6	26.9	152.9	23.10	5.68
2000	60000	20.2	46.5	310.6	15.31	6.68
5000	150000	59.8	90.3	788.1	13.17	8.73
10000	300000	121.1	140.6	1545.0	12.75	10.99
15000	450000	179.4	199.0	2349.1	13.08	11.80
20000	600000	257.8	277.0	3121.4	12.10	11.27

Table 2.1: Performance table for marshalling the Dummy class. Crossover point is 250 objects. Timings are in milliseconds.

AMD Athlon XP 1700+ machine with 900MB memory. The timings are in milliseconds. We use HotSpot (in the default client setting) as the Java Virtual Machine, which is the most popular JVM. When running a test, we first marshal a substantial number of objects to give the virtual machine time to *warm up*. During this time, the JVM loads classes and performs just-in-time optimization. Our experience has shown that this approach gives more consistent results. After warming up the JVM, we begin the test. We create a certain number of serializable objects, then pass the objects to the OOS's and measure the time spent. We call this *a benchmark*. After a benchmark is done, we discard the objects and OOS's —together with the hashtables they contain— and run another benchmark with a different number of objects. Thus, each benchmark begins with the Jumbo API and OOS's loaded and optimized, the specialized marshallers not generated. In the tables below, each row represents a benchmark. During our tests, we never write objects to files, they are always written to in-memory streams.

2.3.1 Homogeneous and Near-homogeneous Data

Table 2.1 gives the results for marshalling objects of the Dummy class:

```
public class Dummy implements Serializable {
  Simple simple1;
  Simple simple2;
  int id;
}

public class Simple implements Serializable {
  int id;
}
```

The “Jumbo OOS” column does not include the run-time compilation cost, but “Jumbo + compilation” does. We have shown timings for marshalling 1000 to 20000 objects. The “Bytes written” column gives the size of the data written to the output stream. Jumbo OOS is at least 12 times faster than Kaffe OOS, when run-time generation cost is not in-

Number of lists	Bytes written	Jumbo OOS	Jumbo + compilation	Kaffe OOS	$\frac{\text{Kaffe}}{\text{Jumbo}}$	$\frac{\text{Kaffe}}{\text{Jumbo+comp.}}$
10	19363	6.7	48.9	145.0	21.42	2.96
50	84479	45.1	71.9	723.9	16.04	10.06
100	186877	107.7	131.3	1496.6	13.88	11.39
150	246075	115.8	135.4	2145.9	18.52	15.84
200	352161	144.6	174.4	2896.0	20.02	16.60

Table 2.2: Performance table for linked-lists of Dummy objects. Each list has fifty nodes. Timings are in milliseconds.

Number of objects	Bytes written	Jumbo OOS	Jumbo + compilation	Kaffe OOS	$\frac{\text{Kaffe}}{\text{Jumbo}}$	$\frac{\text{Kaffe}}{\text{Jumbo+comp.}}$
1000	30136	9.9	55.3	154.7	15.55	2.80
2000	73048	26.0	64.2	343.7	13.18	5.35
5000	174812	76.0	103.5	852.2	11.19	8.23
10000	334320	136.3	167.2	1637.0	12.00	9.79
15000	494312	196.9	217.0	2463.6	12.51	11.35

Table 2.3: Performance table for Dummy objects, allowing the fields to be either Simple or SimpleChild. Crossover point is 280 objects. Timings are in milliseconds.

cluded.The crossover points we give were determined by direct observation, not by interpolation from the presented data. We have omitted the timings for small data sets.

In our next test, we marshal linked-lists of Dummy nodes (same as Node class, but with data of type Dummy). Each linked list has 50 nodes. Jumbo OOS is up to 20 times faster than Kaffe OOS in this test. (See Table 2.2.)

Inheritance affects the cost of marshalling because it requires that we test the type of each field and not simply call the marshaller for the declared type of the field[3]. In the previous benchmarks, we did not marshal any objects whose classes had subclasses; thus, the runtime type of every marshalled object was the same as its compile-time type, and, in particular, the one-element cache always held the right class. For the next benchmark (Table 2.3), we marshal Dummy objects, but allow the fields of type Simple to contain either a Simple or a SimpleChild object, determined randomly. The SimpleChild class is shown below.

```
public class SimpleChild extends Simple{
  int otherId;
}
```

[3]Remember that to eliminate some hashtable lookups, we associate a one-element cache with each field. See Section 2.2.

Number of objects	Bytes written	Jumbo OOS	Jumbo + compilation	Kaffe OOS	$\frac{\text{Kaffe}}{\text{Jumbo}}$	$\frac{\text{Kaffe}}{\text{Jumbo+comp.}}$
13210	128140	76.5	1504.1	1830.0	23.92	1.22
39630	372578	239.5	1690.9	5486.0	22.90	3.24
66050	617016	368.2	1837.9	9248.0	25.11	5.03
92470	861454	524.3	1899.5	12789.2	24.39	6.73
118890	1105892	657.4	2065.5	16499.7	25.09	7.99

Table 2.4: Performance table for heterogeneous data. The objects come from a total of 66 classes. Timings are in milliseconds.

2.3.2 Non-homogeneous Data

Data commonly consist of many objects of a variety of classes. This has a significant effect on the performance of our code because it implies a lot more classes being generated and therefore a lot more program generation time. In this section we examine the behaviour of Jumbo OOS on such data.

For this purpose, we serialize `Code` objects. `Code` is a Jumbo class that represents the *partially* compiled version of a program fragment. When it receives information about the usage context of the fragment it represents, it outputs the virtual machine code corresponding to the fragment. A `Code` object has a tree-like structure where the subtrees are the `Code` objects that represent subfragments. In total, the `Code` objects indirectly touch 13210 objects, from 66 classes; 127 kilobytes were written to the stream. The timings are given in Table 2.4. We start by marshalling just one `Code` object, and increment by two on each row (i.e. marshal the object two more times than on the previous row). In this test, Jumbo OOS is faster than Kaffe OOS by approximately 25 times, when the cost of program generation is not counted. However, when code generation time is counted, the improvement relative to the Kaffe OOS goes down to about 1.22 in the worst case. The speed-up will approach 25 as the size of the data set increases, but it only achieves an eight-fold increase on the largest data set we tried.

The generated code shows much less speed-up than for the homogeneous case. Recall that the crossover point when marshalling `Dummy` objects was about 250 objects; now it is about 10500 objects. The problem, of course, is that we are generating code for many classes that have a small number of instances. We discuss this issue in the next section.

2.4 Just-in-time Program Generation

When marshalling heterogeneous data like `Code`, many classes are represented by only a few objects, and the cost of generating the marshalling code for those classes is not repaid. Our analysis of the test with heterogeneous data showed that only 14 out of the 66 classes allocated more than 250 objects. (Recall that, for `Dummy` objects, the crossover point was

Threshold	Number of objects	Jumbo + compilation	Kaffe OOS	Kaffe / Jumbo+comp.
20	13210	659.5	1861.2	2.82
40	13210	576.1	1854.4	3.21
60	13210	527.6	1871.3	3.54
80	13210	521.1	1871.8	3.59
100	13210	482.9	1872.0	3.87
120	13210	525.1	1869.8	3.56
140	13210	515.6	1869.9	3.62
160	13210	541.7	1905.7	3.51
180	13210	551.7	1855.4	3.36
200	13210	550.8	1869.8	3.39
240	13210	562.3	1869.5	3.32
300	13210	592.8	1871.0	3.15
400	13210	706.5	1868.1	2.64

Table 2.5: Performance comparison when threshold value is used. Marshallers are generated only for classes known beforehand to have more than the previously given threshold number of instances. Timings are in milliseconds.

250 objects.) Clearly, the remaining 52 classes will create a significant drag on the overall marshalling process.

To test the hypothesis that avoiding code generation for classes with few objects will yield better results, we ran a set of tests using varying *threshold* values: For each threshold value, we generated code only for those classes which produce at least that many objects in the benchmark. This depends upon our having counted the number of objects for each class beforehand, so this does not represent a viable implementation strategy; we are only attempting to prove our hypothesis. We see in Table 2.5 that at a threshold value of 100, the generated code produces nearly a four times speedup over Kaffe OOS (compared to 1.22 fold speedup when all marshallers are generated). Note that, even at the optimal threshold value of 100, the speedup we can obtain in this situation is much less than we did with the homogeneous collections, because (1) the cost of run-time compilation is great due to the large number of classes and (2) many objects are marshalled by non-generated code.

In this experiment, the number of instances of each class was known prior to marshalling. What shall we do when we do not know that? The situation is similar to JIT compilation [Adv]. HotSpot keeps track of method calls and when a method is called a certain number of times, it is optimized.

Following this idea, our second version of the marshaller counts the number of objects marshalled. Once it has reached the threshold value, it generates specialized code and uses that for subsequent objects of the class. Note that this version will be slower than the previous one, because all objects marshalled prior to reaching the threshold value are marshalled by non-generated code. The results are shown in Table 2.6. Here, we do not

Threshold	Number of objects	Jumbo + compilation	Kaffe OOS	$\frac{\text{Kaffe}}{\text{Jumbo+comp.}}$
20	13210	920.3	1851.6	2.01
40	13210	781.3	1834.6	2.34
60	13210	669.5	1851.0	2.76
80	13210	634.5	1848.5	2.91
100	13210	585.6	1854.4	3.16
120	13210	615.3	1851.0	3.00
140	13210	594.8	1847.9	3.10
160	13210	604.9	1907.9	3.15
180	13210	606.9	1832.7	3.01
200	13210	610.7	1845.6	3.02
240	13210	611.4	1847.3	3.02
300	13210	629.8	1851.0	2.93
400	13210	732.5	1852.9	2.52

Table 2.6: Marshalling 13210 objects, with different threshold values. Number of instances of classes is not known beforehand. Timings are in milliseconds.

Number of objects	Bytes written	Jumbo + compilation	Kaffe OOS	$\frac{\text{Kaffe}}{\text{Jumbo+comp.}}$
1000	30000	37.0	151.7	4.09
2000	60000	51.9	316.1	6.08
5000	150000	90.2	802.6	8.89
10000	300000	149.9	1592.3	10.61
15000	450000	212.0	2395.2	11.29

Table 2.7: Performance when marshalling `Dummy` objects with threshold value of 100. Timings are in milliseconds.

reach the previous speedup factor, but instead reach 3.16 (again with 100 as the threshold).

Our final version of the marshaller uses the just-in-time idea with a threshold value of 100. We ask our last question: Does this version extract a significant penalty when marshalling *homogeneous* data? Table 2.7 shows the timings for this version of the marshaller, when marshalling collections of `Dummy` objects. This table is comparable to Table 2.1, and it shows that the JIT approach has almost no effect on performance for large homogeneous data sets.

It should be noted that if we have the opportunity to do off-line program generation, using specialized marshallers is the obvious decision, because we wouldn't have the run-time compilation cost. In this case, we would generate the specialized marshallers once before run-time and then at run-time we'd get the benefit of using them. Unfortunately off-line compilation is not always possible.

Number of objects	Bytes written	Jumbo + compilation	Sun OOS	$\frac{\text{Jumbo+comp.}}{\text{Sun}}$
1000	30000	45.1	11.1	4.06
2000	60000	55.9	15.8	3.51
5000	150000	79.1	42.3	1.86
10000	300000	123.3	85.1	1.44
15000	450000	156.2	131.8	1.18
20000	600000	201.5	187.5	1.07

Table 2.8: Performance of Jumbo OOS vs. Sun OOS. Marshalling Dummy objects, program generation cost included, threshold value 100, incorporating lightweight hashtable. Timings are in milliseconds.

2.5 Sun's ObjectOutputStream

The aim of this chapter is to show that RTPG using Jumbo is an easy and effective way to achieve higher performance. In this, we have reached the end of our exposition. However, there are some loose ends to tie up. In particular, the reader may wonder how our code stacks up against the marshalling code that is delivered with HotSpot, which, as we have mentioned, uses unsafe, native code. (To be more specific, it uses the sun.misc.Unsafe class to access arbitrary memory addresses.) Another natural question is whether the kind of program generation we have done can be applied *to* the HotSpot code.

In Table 2.8, we show the result of a test marshalling Dummy objects again, comparing Jumbo OOS (with threshold value of 100) to Sun OOS. To be fair to Jumbo OOS, we note that, in addition to using native methods, Sun OOS uses a custom, lightweight hashtable implementation, which is considerably more efficient than the standard implementation in this context. We incorporated this hashtable implementation into our code, too. In this test, Jumbo OOS is about 4 times slower than the Sun OOS on the small data set with 1000 objects, and only 7% slower than Sun OOS on the largest data set with 20,000 objects.

So, to summarize, while remaining entirely in the realm of verifiable Java code, we have obtained an implementation that can marshal large data sets nearly as fast as Sun's implementation.

Finally, we have experimented with applying RTPG to Sun OOS. We implemented Jumbo OOS and ProgGen using the same principles we discussed in Section 2.1 and 2.2, but based on Sun OOS instead of Kaffe OOS. (Although Sun OOS achieves its speed from using native methods in critical places, much of it is written in Java.) Comparing this version of Jumbo OOS to Sun OOS, we achieve speedups as high as 30% when run-time compilation cost is excluded. However, the crossover point is around 12,000 objects for homogeneous data sets. In conclusion, with program generation, we are able to perform very close to the code with native methods while staying in the realm of managed code. We are also able to improve the performance of code that contains native methods.

2.6 Related Work

Most work on optimizing marshalling is not directly comparable to ours in that the goal is not to optimize the existing, generic marshaller, but to create more efficient marshallers for special cases. For example, Nester et al. [NPH99] require that classes that are to be marshalled must provide their own writeObject method, and also depart from the Sun serialization format in other ways which are valid in their environment, but not in general.

Manta [MvNV+01] and Ibis [vNMW+05] both use run-time code generation to produce specialized marshallers at run time. Their methods are different from ours: In Manta, a compiler is invoked at run time (again requiring that all computers have a specified set-up in order to use their system); in Ibis, a specially built program generator producing JVM code has been written just to generate serializers.

Serialization is used as an example in two papers on RTPG systems that we know of. Neverov and Roe give the definition of a multi-stage language called Metaphor [NR04], in which, in principle, serialization code can be generated in a *type-safe* manner. However, they do not tackle the entire Java serialization specification, and it is not clear whether their techniques could scale to this case. Consel et al. [CLM04] discuss marshalling for C, using the C-based Tempo system.

2.7 Conclusions

In this chapter we have applied runtime program generation (RTPG) to marshalling. We have shown how to generate marshallers specialized for specific types of objects. We have performed experiments with marshalling homogeneous, near-homogeneous and heterogeneous data. Serialization of heterogeneous data illustrated a problem of RTPG. Namely, some generated programs cannot compensate for their generation costs. We have shown that just-in-time generation can successfully help avoid generation of programs that are not likely to pay-off. Our empirical results also suggest that just-in-time generation does not pose a significant overhead on serialization of homogeneous data. Hence, we conclude that just-in-time generation can be considered as an effective method to reduce the cost of runtime generation.

We have based our implementation on the serialization class provided by the Kaffe JVM. We have obtained significant speedup when compared to this code. For some data sets we nearly reached the speed of Sun's object serializer, which extensively uses unsafe native code, while staying entirely in the realm of verifiable byte code.

In all of our experiments, we serialized objects into in-memory streams. However, it is common in applications that do marshalling to have I/O operations which may dominate the time spent. It would therefore be interesting to experiment with the ideas we discussed here in the context of CPU-bound applications. Computations where availability of some runtime information allows getting rid of condition checks, jumps, and memory accesses, similar to the case with marshalling, can potentially greatly benefit from PG. We plan to look for such applications in the domains such as scientific computing and graphics.

Chapter 3

Source-level Rewriting of Staged Programs

In this chapter we again focus on the first challenge of program generation: the cost of generation. Chapter 2 took the approach of avoiding unnecessary runtime compilation for this problem. We now investigate how we can reduce the cost of generation.

We start by making an observation. In program generation, fragments are composed at runtime to form the final program, which is then compiled for execution. Even though we cannot know the generated program at compile-time, we do have access to individual fragments. This brings the question: Can we take advantage of these fragments to optimize the compilation process of the generated program? Consider the case of a quoted class definition with a hole where a method should go. We can possibly compile the existing methods of the class at compile time, and then combine the result at runtime with the missing method to get the entire class compiled. In its essence this is a partial evaluation problem. The compiler has two inputs — the quoted class and the method — of which only the class is known at compile time. It is quite plausible that we might apply the compiler to the class and obtain a "residual compiler" that will receive the method and complete the compilation at run time. We can take a symmetric look at the problem as well: apply the compiler to the method to obtain a residual compiler, which will later receive the surrounding context of the method to finish up the compilation. However, partially evaluating an ordinary compiler is problematic because (1) it expects a compilation unit (e.g. a class or an interface in Java); anything smaller would be meaningless to the compiler; (2) even when the compiler is fed with a compilation unit (with holes), it would be practically very difficult, if not impossible, to partially evaluate it. Recall that we are advocating generality of program generation. Putting restrictions may provide opportunities for optimization, however our goal is to preserve generality of program generation. To overcome these problems, we take advantage of Jumbo's *compositional* structure. Compositionality means that even the smallest piece of syntax is meaningful. We take these "meanings" and apply source-level transformations to optimize them.

Our contributions in this chapter are two-fold:

- We show that source-level transformations applied at compile-time help to optimize runtime compilation.
- We show why restructuring the compiler into a more functional (i.e. side-effect free)

style makes it more susceptible to transformations.

The chapter is organized as follows: Section 3.1 explains in more detail what compositional compilation means. This is followed in Section 3.2 by the use of compositionality in Jumbo. Section 3.3 describes the analyses and transformations we have implemented and Section 3.4 gives examples and timing results. In Section 3.5, we discuss some of the difficulties presented by Java which have limited our success in optimization. Section 3.6 briefly discusses related work. We finally give our conclusions in Section 3.7.

A version of the work presented in this chapter has been published in GPCE '05 [KAM05].

3.1 Compositional Compilation

Restricting generality of program generation may help a PG system to promote safety and efficiency at the expense of constraining programmer's ability to structure the program-generating process. We advocate generality. Consider the following cases. Is it legal to fill the hole in ⟨int m () {`(hole) return x;}⟩ with the declaration ⟨int x=10;⟩? How about filling ⟨if (y==x) `(hole) else ... ⟩ with ⟨break L;⟩? Is the position of the hole in this fragment legal: ⟨try { ... } catch `(hole) { ... }⟩? Can the hole in ⟨`(hole) class C { ... }⟩ be filled with ⟨import java.util.*;⟩? We would like to have a system that gives the answer "yes" to all of these questions. Jumbo is a program generation system that can do this.

This high degree of generality is achieved by using a *compositional* compiler that gives a "meaning" to any node in the abstract syntax tree (AST) of a program. This provides the ability to divide up the program into almost arbitrary fragments. It also makes it possible to use the same compiler both for compile-time and runtime compilation; the back-end of the compositional compiler serves as the code-generating engine for program generation.

In compositional compilation, the "meaning" given to a fragment is an intermediate representation called *Code* [KCC00b]. The *Code* value of a compound fragment is a function solely of the *Code* values of its subfragments. Filling in a hole is performed by placing the *Code* value of a fragment inside another *Code* value. The compiler contains functions to translate an AST to its *Code* value. Three examples are given below (the parameter flags encodes the modifiers such as public, static, etc.):

```
Code makeIfThen (Code cond, Code truebranch)
Code makeVariable(int flags, Type type, String name)
Code makeClass(int flags, String name, String supername,
               StringList implementees, CodeList members)
```

This is the difference between a compositional and a conventional compilation structure: Instead of creating an AST and then generating machine code while traversing it, the abstract syntax operators themselves are converted to code to compile that syntactic construct. A question that may arise with this definition is how to analyze programs. Converting a syntactic construct to not only a function that is going to produce the machine-level code, but also to a function (or a collection of functions) that is going to produce the analysis result overcomes this problem.

A preprocessing step translates quoted fragments to abstract syntax operators. For example,

```
Code safePointer (Code ptr, Code computation) {
   return ⟨ if (`(ptr) == null)
                throw error();
            else `(computation) ⟩;
}
```

becomes (0 is the code for binary operator "==")

```
Code safePointer (Code ptr, Code computation) {
   return makeStatements(
            makeIfThenElse(
              makeBinOp(0, ptr, makeNullConstant()),
              makeThrow(makeSelfInvocation("error", new List())),
              computation));
}
```

This program is now statically compiled — that is, as an ordinary program. The calls to the abstract syntax operations are part of the program and will be elaborated at run time, after the holes have been filled in. Note that holes are handled with no special effort — they are just expressions within a larger expression which do not happen to be explicit calls to abstract syntax operations. In particular, at run time, *Code* values will be provided for the arguments to safePointer, and a *Code* value that represents the if-statement with the holes filled in will be returned. Eventually, this *Code* value will be placed inside the *Code* value for a compilation unit, and be ready for the final step of compilation — generating machine code (e.g. Java .class files containing JVM code). In Jumbo, the method void generate(), provided as part of the *Code* value, performs this final step. Alternatively, Object create(String classname) calls generate, and then loads the class file and returns an object of the class. generate is for off-line program generation, and create is for run-time program generation.

3.2 Structure of Jumbo

There are many choices for the *Code* type [KCC00a, KCC00b]. A naive version would use AST's as *Code* and generate would do all the work. Jumbo aims to leave as little work to generate as possible. This is done by making each *Code* value a function taking the compilation context (or "environment") to JVM code. This is how compositionality is achieved in defining abstract meanings of programs in denotational semantics [Sto77]. In Java, the situation is a bit more complicated, but the idea follows in general. *Code* values are represented by objects having a single method, plus some additional information:

$$Code = ExportedDefinitions \times (Environment \rightarrow ClosedCode)$$

$$ExportedDefinitions = (ClassInfo + MethodInfo + FieldInfo)$$

$$Environment = \text{stack of } (ClassInfo + MethodInfo + LocalInfo)$$

$$ClosedCode = JVM\ code \times integer \times integer \times VarDecls \times Value$$

The first component of *Code* is the declarations exported from the code fragment. The second is the function we have been referring to above, which we call *eval*; it does the actual translation to JVM code. *ExportedDefinitions* are the declarations that are in scope outside of this fragment. Based on the exported declarations of a class's members, the class can create a fairly complete record of its contents, and that record (a *ClassInfo*) will be its exported declaration. The *eval* method is given an environment containing all enclosing classes, methods, and variables, and then generates code. The two integers in *ClosedCode* give the next available location for local variables and the gensym seed, needed to assign unique names to anonymous classes. The *VarDecls* value carries the local variable declarations of that code fragment. The *Value* field gives the constant value of an expression, if it has one; the Java language definition [GJS96, §15.27] requires this.

We believe this definition of *Code* is as compact as possible. We now explain briefly why this definition works. In Java, names fall under two scope rules: names defined within a method — local variables and inner classes — are in scope in statements that follow the declaration ("left-to-right" scope), while names defined in a class — fields and inner classes — are in scope everywhere within the class (with the exception that fields again have a "left-to-right" scope among themselves and are not visible in their own initializers). The exported definitions in *Code* are used to create the latter part of the environment; the environment passed into the *eval* function of the methods of a class contains all the fields and inner classes of that class. Names with left-to-right scope are passed along in the environment from one statement to the next, using the *VarDecls* in *ClosedCode*. Thus, the *eval* function for each statement gets an environment containing all the names in scope at that statement. (As a technical point, this definition is actually a little bit too parsimonious, in that it does not allow a proper treatment of free variables in inner classes. The rule about inner classes is that each variable captured by an inner class becomes a read-only field of the inner class, and the constructors of the inner class must assign the variable to its corresponding field. The question is, how do we know which variables are actually used in an inner class? This information does not come from the exported definitions of the inner class, since references are not definitions, nor is it passed "left-to-right." We finesse this problem by assuming that all variables in scope in an inner class are referenced in that class. This gives a correct, but obviously non-optimal, implementation of inner classes.)

```
Code makeMethod(Type type, String name, List args, Code body) {
  return new Code() {
    DeclarationList getDecls() {
      return new DeclarationList(decl of the method);
    }
    ClosedCode eval(Environment env) {
      env = env.add(args);
      ClosedCode cc = body.eval(env);
      ...
    }
  };
}

Code makeField(Type type, String name, Code init) {
  return new Code() {
    DeclarationList getDecls() {
      return new DeclarationList(decl of the field);
    }
    ClosedCode eval(Environment env) {
      ClosedCode cc = init.eval(env);
      ...
    }
  };
}
```

Figure 3.1: The makeMethod and makeField methods of the compiler that construct Code objects for a method and a field definition, respectively. The code here is simplified to eliminate parts that are irrelevant to our presentation in this chapter.

In the implementation, *Code* is represented with an abstract class named Code that has two methods: DeclarationList getDecls(), and ClosedCode eval(Environment). Each method corresponds to an element in the definition of *Code*. Let us now look at an example to see more details of the implementation. Suppose we have the following quoted fragment that contains a field declaration and a method that refers to the field.

```
⟨ int getX() {
    return x;
  }
  int x = 0;
⟩
```

This fragment would be preprocessed to a list that contains two Code objects as created by the following calls (visibility tags of the method and the field are ignored):

```
makeMethod(Type.INT, "getX", new List(),
           makeReturnStmt(makeVarAccess("x")))

makeField(Type.INT, "x", makeIntLiteral(0))
```

```
Code makeClass(int flags, String name, String supername,
               StringList implementees, CodeList members) {
  return new Code() {
    DeclarationList getDecls() {
      return new DeclarationList(decl of the class);
    }
    ClosedCode eval(Environment env) {
      // first pass: collect declarations
      foreach member in members
        env = env.add(member.getDecls());

      // second pass: emit JVM code
      foreach member in members
        ...
        ClosedCode cc = member.eval(env);
        ...
    }
  };
}
```

Figure 3.2: The makeClass method of the compiler that construct Code object for a class definition. The code here is simplified to eliminate irrelevant parts.

The methods makeMethod and makeField are implemented as in Figure 3.1 (greatly simplified to eliminate irrelevant parts). The implementation makes use of anonymous classes that extend the abstract Code class. The compositional structure can be seen inside an eval method where the eval of the subcomponent is invoked. To improve the efficiency of the execution of an eval, we inline the calls to eval methods of the subcomponents and perform traditional optimizations on the code. Inlining the invocation body.eval(env) inside makeMethod will reveal the call to eval of the return statement, which contains a reference to the field x. However, because we cannot know the contents of the environment that comes as an argument to eval of the method, there is no way to know that the referred variable is in fact the field x. On the other hand, if we had the information that the declaration defined by the field exists in the environment that will be passed to the eval of the method, we could utilize more optimizations. We would have this information if the method and the field definition were given inside a class, because a compilation unit first adds declarations into an environment before calling eval of its subcomponents. The implementation of makeClass, shown in Figure 3.2, illustrates this. However, compilation units are not often available. This prevents us from making the connection between getDecls and eval. Therefore, putting as much information into a single pass and reducing the number of passes is important to reveal opportunities for optimization as much as possible. The definition of *Code* aims at this.

Achieving the final definition of *Code* required several revisions from its previous definition, originally implemented by Lars Clausen [Cla04]. When compiling a program, Jumbo performs "passes" on the code. Jumbo originally had four passes. The generate method contained the following piece of code where the four passes of the compiler (defineClasses, defineSupers, defineMembers, and eval) can be seen:

```
ClassEnvironment e = defineClasses(new ClassEnvironment());
eval(defineMembers(defineSupers(new Environment(e))));
```

The methods defineClasses, defineSupers, defineMembers, and eval were defined in the Code class. Every pass collects important information that is used in the subsequent passes. Not having access to the call to generate makes us lose the connection between different passes and results in poor optimization as discussed above. With the new definition, we decrease the number of passes to two: the first one, getDecls, collects exported definitions, and the second one, eval, emits JVM code using the information collected in the first pass.

This new definition of *Code* required non-trivial refactoring of the Jumbo compiler. During this restructuring, we also used a "functional style" implementation. We used, for instance, final fields (i.e. fields that cannot be reassigned once initialized) whenever possible and preferred immutable linked lists over arrays. This is important to obtain better optimizations, because hard-to-prove properties such as escaping and aliasing prevent progress in the presence of side-effects.

To summarize, our task comes down to this: In a Jumbo program, sections of quoted code become expressions of type Code. At run time, these expressions will be evaluated, producing a Code object whose getDecls and eval functions will then be invoked. We wish to optimize this entire process, but mainly the eval function of each *Code* value, since this is where most of the compilation occurs. Optimization is done by applying transformations on the source code.

3.3 Source-level Optimization of Java

In this section, we describe the optimizations we apply. These take the form of source-level transformations, including method inlining, constant propagation, and various simplifications.

In this experiment, these optimizations were not all applied automatically. A number of transformations are "contractive" — simply put, they never make things worse — and they are applied repeatedly in a "clean up" process. Others — such as inlining — are potentially dangerous, in that they can lead to code expansion, and the system must be manually told to perform them. (A user interface highlights all inlinable methods and constructors, and the user clicks on the method name to inline it.)

The transformations are mainly standard and will be described only briefly. We emphasize that all are valid transformations in Java. The idea is not to build an optimizer specific to our compiler, but to pick the analyses and transformations with the knowledge of their intended use. It would also be interesting to see if all the transformations, including function inlining and loop unrolling, could be automated using heuristics based on the knowledge about the internals of the compiler. This was done in a prototype compiler as part of the author's MS thesis [AK05, Akt05]. Several of the analyses and transformation described here were originally implemented by Lars Clausen [Cla04].

3.3.1 Normalization

To reduce the number of cases that need to be handled by the analyses and transformations, the code is first normalized. There are three main parts of the normalization step:

FQCN: Converts every name to its fully qualified version. For instance, a field access `x` becomes `this.x`, and a field declaration `Code c;` becomes `uiuc.Jumbo.Compiler.Code c;`.

For-While: Converts for-loops and do-loops to while-loops.

Flattening: Breaks complex expressions into simpler expressions. For instance, after this step, all the arguments going into a method call will be simple variables.

3.3.2 Transformations

The following rewrites are applied after the normalization step. All must be applied "manually" — that is, by explicitly requesting the rewriting engine to apply them. However, Cleanup incorporates many of them in a fixpoint iteration; those are not normally invoked manually.

Inlining: Inlines a method invocation. Replaces return-statements of the inlined method with break-statements.

WhileUnroll: Unrolls the first iteration of a specified while loop.

AnonClassConvert: Converts anonymous classes to non-anonymous inner classes.

Unflatten: Transforms the flattened program to a form that is more readable.

ConstructorInlining: Most of our transformations and analyses are strictly intra-procedural. This makes inlining very important for exposing opportunities for optimization. During our experiments, we noticed that some object creations prevented several optimizations from taking place because object references were escaping into constructors. To propagate information better, we decided to inline constructors as well. However, constructors cannot be inlined like methods, because there is no notation to create an uninitialized object in Java; this is an implicit effect of each constructor. (If we were optimizing JVM code instead of source, this would not be a problem.) We might try to use the zero-argument constructor for this purpose, but it might have an explicit definition that conflicts with the definition of the constructor we are attempting to inline. We solve this problem by adding *annotations*, containing the statements of the constructor, to object creation sites. Other rewriters then see the constructor code as though it was just an inlined method. The constructor itself, which resides in a separate class, cannot be optimized, but values propagated out of it can be used in the calling program. The annotations must be removed before the optimized program is written; for this reason, the annotations must have the property that they can be removed at any time and leave a program with the same meaning as when they were there. Philip Morton gives more details about constructor inlining in his MS thesis [Mor05].

Cleanup: Runs the following rewrites in a fixpoint iteration. Each can be invoked manually, but there is little reason to do so.

Untupling: Extracts a field from a newly created object.

UnusedDecl: Removes declarations that are never used.

UnusedScope: Removes scopes that have no semantic significance.

UnusedDef: Removes variable definitions that are not used.

UnusedReturn: Eliminates assignment of a method call when the assigned variable is not used. The method call must still be executed for its side effects.

IfReduction: Simplifies if-statements whose condition is a constant boolean.

Arithmetic: Simplifies constant-valued arithmetic and logic expressions.

UnusedBreak: Removes break statements that make no difference to the flow.

ConstantPropagation: Moves constant values through local variables.

CollapseSystemCalls: Collapses `intern` and `equals` calls made on `Strings`.

ArrayLength: Replaces `array.length` expressions with the length, if available.

Switch: Reduces constant switch statements to the match.

CopyAssignment: Propagates redundant assignments of variables and literals.

UnusedObject: Removes object creation statements if they are never used and side-effect-free.

FieldValue: Propagates values through object fields assigned directly.

TightenType: Makes types more specific, if possible.

UnusedFieldAssign: Removes unused assignments to fields.

UnreachableCode: Removes code which is indicated to be unreachable by the flow analysis.

ObjectEquality: Replaces (obj1 == obj2) with true, and (obj1 != obj2) with false, if it can determine whether the two objects point to the same location; and vice versa.

PointlessCast: Removes cast expressions where the target of the cast is already of the right type.

WhileReduction: Removes while statements which only have a break as the body and/or false as the condition.

InstanceOf: Attempts to resolve instanceOf expressions.

NullCheck: If it can prove that an object o is not null, then replaces o != null with true and o == null with false; and vice versa.

These rewriters use the information obtained from program analyses. The analyses are **Dominator**, **Flow**, **Use-Def** and **Alias**. The first three are standard. Alias analysis is described in Philip Morton's MS thesis [Mor05].

3.4 Examples

We demonstrate the effect of our optimizations via three examples. The first is a complete (but small) class, without holes. The other two are the classic (in the field of program generation) exponentiation function, and a program to generate finite-state machines, taken from [Kam03].

For each example, we show the original program, with quoted fragments. The latter will be preprocessed away and transformed to calls to abstract syntax operators, as described in Section 3.1. The resulting program is an ordinary Java program that will be compiled into JVM code and executed. At run time, the various *Code* values produced by these expressions will be brought together to form a *Code* value representing a class. A call to generate or create will turn this Code value into a Java .class file. In our examples, we are not executing the generated programs, since we are interested only in code generation time — the code that would be generated in each version is exactly the same. In each test, we let the virtual machine "warm up" — load the Jumbo API, java.lang, and other classes — before executing the programs, then run each test 500 times. Our measurements exclude I/O time for outputting the .class file.

To obtain the optimized versions of the programs, each quoted fragment is optimized, in isolation, after it is preprocessed, using the rewritings described in the Section 3.3.

For each run — original or optimized — we measure the overall time, and we also measure the time spent in the method Class.forName. This method does the run-time look-up for names used but not defined in the program (for example, classes defined in imported packages). It consumes such a large portion of run-time compilation time — more than 50% in most cases — that its effect on speed-up is often substantial. Furthermore, these

	HotSpot		Kaffe		IBM	
	w	w/o	w	w/o	w	w/o
Original	0.46	0.44	0.79	0.76	0.42	0.41
Rewritten	0.40	0.38	0.66	0.64	0.47	0.46
Speed-up	13.0%	13.6%	15.2%	15.8%	-11.9%	-12.2%

Table 3.1: Run-time generation performance for the simple example of Section 3.4.1. The "w" column includes the cost of Class.forName calls; "w/o" column does not. All timings are in seconds.

calls are impossible to eliminate by any compile-time optimization, since the imports must be elaborated on the target machine (i.e. at run time). Since this cost is specific to Java, it is interesting to see what speed-up we would be getting if this cost could be ignored.

The tables in this section have two columns for each of three different Java virtual machines: Sun's HotSpot [Javb], Kaffe [Kaf] (an open source VM), and IBM's production VM [IBM]. For each VM, we give the overall execution time and the execution time excluding forName calls; these are the "w" and "w/o" columns, respectively. The tables have three rows: unoptimized time (labeled "original"), optimized time (labeled "rewritten"), and speed-up ((unoptimized time - optimized time) / unoptimized time).

The timings are in seconds. Tests were run on an AMD Duron 1GHz processor, with 790 MB of memory, running Debian Linux.

3.4.1 Simple Class

To get a feeling of the baseline, we show the results of optimizing a complete, but simple, class. The tests just invoke generate on this code:

```
⟨ public class Temp {
    int x;
    int id() {
      return 12;
    }
  } ⟩
```

When presented with a complete class without holes, the rewriters ought to be able to reduce it to a very efficient form. However, the speedups are not as great as we would hope. (In the case of the IBM JVM, the rewriting actually produced a slow-down.) Reasons for this are discussed in Section 3.5.

3.4.2 Exponent

The exponentiation example, given in Figure 3.3, generates a function that computes x^n for given value of n. Table 3.2 gives the performance of the original and rewritten programs.

```
interface ExpClass
{ public int exponent(int x); }

public class Power {
  public static ExpClass getExp(int n) {
    Code r = ⟨1⟩;
    for(int i = 0; i < n; i++){
      r = ⟨ `(r) * x⟩;
    }
    String cname = "Power"+n;
    Code expcl = ⟨
        public class `(lift(cname)) implements ExpClass {
          public int exponent(int x) {
            return `(r);
          }
        }
    ⟩;
    return (ExpClass)expcl.create(cname);
  }
}
```

Figure 3.3: The generator that produces a specialized exponentiation function.

	HotSpot		Kaffe		IBM	
	w	w/o	w	w/o	w	w/o
Original	2.79	0.98	2.45	1.29	4.55	2.98
Rewritten	2.70	0.89	2.36	1.09	4.19	2.63
Speed-up	3.2%	9.2%	3.7%	15.5%	7.9%	11.7%

Table 3.2: Run-time generation performance for the exponentiation example in Figure 3.3. Timings are in seconds.

3.4.3 FSM

Another application of RTPG is generation of finite state machines (FSM). The example is discussed in [Kam03] and we give the code of the FSM class in Figure 3.4. ArrayMonoList is just a type of list; here it is used to collect all the cases in the switch statement that is the heart of the FSM implementation. Antiquoting an ArrayMonoList splices its contents into the hole. The source code of the other classes (Predicate, Action, Transition, State) can be found at `http://loome.cs.uiuc.edu/Jumbo/examples/FSM`.

The constructor of the FSM class takes a finite-state machine description in the form of an array of states; the client sends the genFSMCode message to that object and then invokes generate on the result. The created class contains a main method that reads a string from the console and runs the client's FSM on it. An FSM is a set of states, and each state is a set of transitions.

```
public class FSM {
  String FSMclassname;
  State[] theFSM;

  FSM (String c, State[] M) { FSMclassname = c;  theFSM = M; }

  Code genFSMCode () {
    ArrayMonoList body = new ArrayMonoList();
    // Each state corresponds to a case in the switch statement
    for (int i=0; i<theFSM.length; i=i+1){
      body.addAll(⟨case `(lift(i)):
                      `(theFSM[i].genStateCode(⟨ch⟩))
                      break; ⟩);
    }
    Code result =⟨
          import java.util.*;

          public class `(lift(FSMclassname)) {
            static void runFSM (StringTokenizer in) {
              int theState = 0;
              while (true) {
                char ch;
                if (!in.hasMoreTokens()) return;
                ch = in.nextToken().charAt(0);
                switch (theState) {
                  `(body)
                  default: return;
                }
              }
              return;
            }
            static void addToBuffer(char ch){ ...  }
            static void emitbuffer(){ ...  }
            public static void main (String[] args) {
              String input = ...; // obtain input from console
              runFSM(new StringTokenizer(input));
            }
          }⟩;
    return result;
  }
}
```

Figure 3.4: The finite-state-machine example [Kam03] used in the experiments.

	HotSpot		Kaffe		IBM	
	w	w/o	w	w/o	w	w/o
Original	13.10	4.93	14.01	8.82	8.92	3.89
Rewritten	12.25	4.76	13.48	7.78	8.37	3.70
Speed-up	6.5%	2.9%	3.9%	11.8%	6.2%	4.9%

Table 3.3: Run-time generation performance for the FSM example.

The definition of a single transition is

```
new Transition(new Predicate1 (), 1, new Action2 ())
```

where

```
class Predicate1 implements Predicate {
  public Code pred (Code ch) {
    return   〈 ('a' <= `(ch) && 'z' >= `(ch))
                || ('A' <= `(ch) && 'Z' >= `(ch) )〉 ;
  }
}

class Action2 implements Action {
  public Code action(int s, Code ch) {
    return 〈addToBuffer(`(ch));〉;
  }
}
```

This transition, upon seeing a letter, goes from its current state to state 1 and puts the letter into the buffer.

Table 3.3 gives the program generation timings for the FSM example. It is notoriously difficult to understand the performance of Java virtual machines, and Table 3.3 is an example. The calls to forName are a large percentage of the execution time on all VMs. Furthermore, these calls are identical in optimized and unoptimized code (recall that we exclude the actual writing of the .class file and its loading into the virtual machine). Yet speed-ups in two cases (i.e. HotSpot and IBM) actually *decline* when forName is discounted. This is because, even though optimizations do not touch this method, it runs faster in the optimized than in the original code. We have, at present, no explanation for this behavior.

3.5 Lessons Learned

Compositional compilation can be applied to any language, yielding a compiler that supports run-time program generation (once the quotation/anti-quotation syntax is added). Each language will present different issues, both in construction of the compiler and in optimizing run-time program generation. Java is in some ways highly suitable for this treatment. Because it has no preprocessor and no optimization pass to speak of, most of the compiler consists of a translator from AST's to low-level code — the process to which compositionality applies most naturally. But in another sense, Java is *too dynamic*; some compilation steps must be performed dynamically that, in other languages, can be performed statically. Obviously, anything that must be done at run time cannot be optimized away. In this section we discuss why we have not gotten better speed-ups.

The major issue blocking rewriting is resolution of class names. The Java definition requires that these names be resolved on the target machine. Thus, for example, the test to determine if a method override is legal — which must be done for every method — cannot be eliminated, because the superclass is available statically only in the rare case when it is defined in the quoted fragment itself.

Similarly, the normalization of class names (conversion of a short class name to a fully qualified class name) for variable, field, and method declarations must be done dynamically. This necessitates that the fields keeping track of type information be mutable: The objects containing those fields are the class and method objects created by getDecls, but normalized class names cannot be filled in until eval is called. Moreover, these objects are returned from getDecls to generate, so unless the fragment being optimized is in a place where the generate call can be inlined — which it usually is not — the class and method information have to be considered to have "escaped." Propagating information through mutable fields of objects that escape is very difficult.

One result is that the optimized code generator still contains type checks which we would initially have expected could be eliminated, such as a check for the validity of the return statement in ⟨int foo() { return 5; }⟩.

Even if the fragment being optimized consists of a complete class, it is possible that the consumer of the fragment will compile it in a larger context: adding import statements, adding sibling classes, or making it an inner class. Not knowing this context causes more class name resolution problems. For example, if an enclosing class contains a field named "java", then "java.lang.Object" represents a series of field lookups, not a fully qualified class name. Having an explicit create or generate call available in the code being optimized resolves this difficulty, because it tells us that the fragment we see will not be placed in any larger context.

We have continually refined our compiler in two ways. One is reducing the number of "passes" — that is, the number of functions in *Code*. The idea is that putting more work in a single pass makes more information available locally; with multiple passes, each called from generate, the connection from one pass to another cannot be inferred except in those cases where we can see the generate call and inline it. As mentioned in Section 3.2, the current structure is as compact as we think is possible.

The other refinement is making the fields in the compiler's classes final. There is a bit more that can be done along these lines.

More broadly, however, Java fundamentally limits optimizations because of the requirement to locate classes dynamically. This entails run-time calls to forName; in one case — the exponentiation example in HotSpot — forName consumes 65% of run-time compilation time. We have also noted above how dynamic class locating has a cascading effect: it requires that certain fields be mutable, which in turn diminishes our ability to statically determine their values.

3.6 Related Work

A compromise of high degree of generality is efficiency of code generation. In particular, being able to fill in a hole with a fragment that can modify the environment (e.g. by declaring a new variable, or defining a class) prevents us from reducing a fragment all the way down to machine code at compile-time. Some systems inherently eliminate this problem by putting restrictions on what can be quoted and/or how the fragments can be combined. Template-based program generation approaches, such as Tempo [CLM04], Cyclone [SGM+03], DynJava [OMY01] and `C [EHK96] construct programs at run-time by combining pre-compiled fragments. They are therefore limited to fragments that generate machine code; declarations cannot be used to fill-in holes because declarations produce no machine code. In general, all the type information in a fragment has to be available to be able to generate a machine-level template for that fragment at compile-time. For instance, in DynJava, the type of every free variable in a fragment has to be given by the programmer. DynJava uses this information to surround a fragment with mock methods and then use the javac compiler to generate bytecode for the fragment. In the presence of high degree of generality as in Jumbo, it is not possible to know all the types; Lars Clausen gives a discussion and an example that covers different possibilities [Cla04, §3.4.1]. We could utilize more optimizations if there were a way to know that a hole does *not* introduce any new bindings. This would make sure that an environment that enters a hole is exactly the same as the one that come out, making it possible to do lookups safely in that outgoing environment.

In PG by partial evaluation systems [TS00, TCLP, MTBS99, CX03, GMP+00] every variable in a fragment has a binding. (These systems in fact possess the "erasure property" — erasing quotation marks leaves a valid program which is equivalent to the original but is not staged.) Thus, they follow ordinary scoping rules for declarations, and the generation process cannot introduce new declarations.

3.7 Conclusions

We have shown how source-level optimizations can improve the performance of a program generation system based on the principle of compositional compilation.

The Jumbo compiler was first publicly released in 2003. We began the current study from (the newest version of) that compiler, but found that compositionality alone was not enough to permit optimization. We rewrote the compiler to be (a) *more* compositional — where the first definition of *Code* contained four functions, the current one has two — and (b) more functional in style, making greater use of final fields. It seems reasonable that, since RTPG can offer very significant performance advantages, the compilers to support it might be written so as to allow for more efficient code generation. We believe that the points we explained in the "Lessons Learned" section will be useful to researchers in writing new RTPG compilers or in revising existing ones. It would be interesting to see the optimizations we cover here be applied on a language less dynamic than Java, such as C.

Chapter 4

Staging Static Analysis of Generated Programs

In this chapter we again address the first problem of program generation: the cost of runtime generation. Generation of a program requires running several analyses to check that the program is compilable or to optimize it. The question we ask is this: Can we reduce the time we spend for these analyses for more efficient runtime generation? At compile time, we have access to individual program fragments, however, we cannot know how these fragments will be put together to form the final program (on which analyses will be run). Therefore, in general, it is not possible to compute the result of an analysis completely at compile-time. Nevertheless, we may be able to perform a portion of the analysis at compile-time by analyzing the individual fragments. In particular we are given a program $P[\bullet, \ldots, \bullet]$ with holes, and a collection of plugs $Q_1, \ldots, Q_n$. We want to find the result of some static analysis when applied to $P[Q_1, \ldots, Q_n]$. We can preprocess P and the Q_i, and then combine the results at run time to produce the same analysis result. This is the topic of this chapter.

We present a technique that addresses this problem by splitting the analysis of runtime-generated programs into two stages: compile-time and runtime. This is done by means of a compositional framework for defining program analyses. The framework leads directly to a method of starting the analysis of incomplete programs at compile time; the residual work to be done at runtime may be much less costly than the full analysis. The ability to stage analyses depends upon finding an accurate representation for the dataflow functions; we present such representations for several analyses. Our framework is defined on abstract syntax trees (AST), because program fragments appear as AST's. There is no fundamental reason why we could not have used control flow graphs (CFG) with holes instead of AST's; CFG's and AST's are just representations of programs. Michael Katelman [Kat06] gives formal proofs showing that our AST-based dataflow framework is correct with respect to the traditional CFG-based approach.

The following are the contributions of this chapter.

- Definition of frameworks for forward and backward analysis of abstract syntax trees (AST), including break statements. We show how analyses instantiated from these frameworks can be staged to reduce the time spent for analyzing a generated program at runtime. Staging requires that dataflow functions be represented "adequately." We provide formal proofs stating that staging gives the same result as the original (unstaged) analysis.

- Definition of representations for several dataflow problems.
- Experimental results showing the performance benefits obtained from staging.

The chapter is outlined as follows: In Section 4.1 we present the framework for forward analysis, using *uninitialized variables* as a simple analysis example. We also discuss how the framework allows for efficient staging of analyses. In Section 4.2 we present definitions of several analyses that use the framework. Section 4.3 presents the backward analysis framework. Section 4.4 gives performance results for various analyses and benchmark programs. The proofs of the main theorems stated in this chapter are available in the Appendix.

The contents of this chapter have been published at GPCE 2006 [KAK06].

4.1 Framework for Forward Analysis

Our framework differs from the standard one [ASU86] in that it analyzes abstract syntax trees, not control-flow graphs (CFGs). Since program fragments appear as ASTs, this is the natural unit of analysis for our purposes. Note that we are considering only intraprocedural analysis in this work.

In AST-based static analysis, as in standard control flow graph-based analysis, each node is, in the end, assigned a value from a lattice $Data$ of dataflow values. However, in the AST case, the assignment is performed by a traversal of the tree (rather than by a worklist algorithm), possibly including multiple traversals of some subtrees. Thus, each node has input data (received from its neighbor to the left or right, depending upon whether we are considering a forward or backward analysis) and output data. A key difference is that the AST contains nodes that represent entire subtrees, so that the calculation of output data from input data may be the composition of many smaller calculations. Whereas in a CFG, the function from input data to output data given by any one node is relatively small, in an AST it can be very large. (AST's do, of course, contain those "small nodes" as well; they just have *more* nodes overall.)

Given a (hole-free) subtree, taken out of context, we cannot say what its value is because we do not know its input data. We do know the function from $Data$ to $Data$ that it represents. Now suppose we have representations for functions that arise in a particular analysis. Then we can handle staging of the analysis like this: For all AST's, calculate this function for every *maximal hole-free subtree*. This leaves a prefix of the original AST, with some subtrees pruned and replaced by function representations. (For hole-free plugs, the entire tree is replaced by its function representation.) At run time, the code-generating

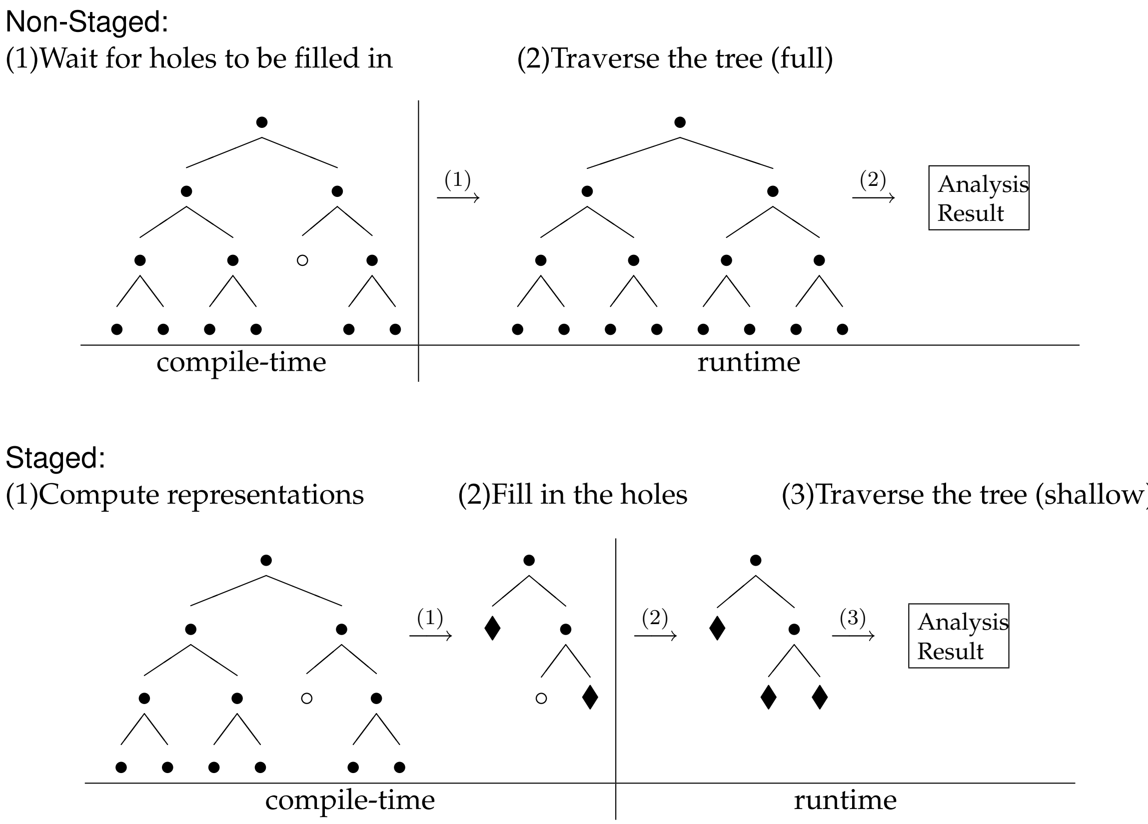

Figure 4.1: Staging a data flow analysis. • is a regular AST node, ◦ is a hole, and ♦ is a representation.

code associated with each fragment [KCJ03] is accompanied by the fragment's representation tree. When the fragments are combined to form the entire program, the static analysis can be performed on the combined tree. Time is saved because there is no need to traverse the program's entire AST, and also because there may be optimizations applicable to the function representations.

The staging process is illustrated in Figure 4.1. For this process to work, we need the dataflow framework to have a key property: compositionality. Being compositional means that even the smallest piece of AST has a meaning from the point of view of dataflow analysis. This is where our framework is distinguished from existing AST-based dataflow analyses. These frameworks either take AST's as a synonym of CFG's and use a worklist algorithm to compute the dataflow information (e.g. [Muc97, §8.4]) — which is not compositional, or they assume a simply-structured language with no control-flow-changing statements, such as `break` (e.g. [Muc97, §8.7]). Our framework is able to give meanings to ASTs with non-local control flow; e.g. `break L`.

The "meaning" given to an AST by the framework is nothing but the dataflow function defined by that AST. A question that naturally arises at this point is how to summarize

$$
\begin{aligned}
&e \in \mathrm{Exp} \\
&x \in \mathrm{Var} \\
&\ell \in \mathrm{Label} \\
&P \in \mathrm{Pgm} ::= x = e \mid \mathsf{skip} \mid \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2 \mid P_1; P_2 \\
&\qquad\qquad\quad \mid \mathsf{while}(e)\ \mathsf{do}\ P \mid \ell : P \mid \mathsf{break}\ \ell
\end{aligned}
$$

Figure 4.2: The language treated in this chapter.

these functions efficiently and precisely (i.e. without loss of information), so that these summaries can be used as representations for AST's. Note that a naïve approach would be to use the AST itself as the summary of its dataflow function. This approach obviously cannot yield any efficiency gains. In this chapter, we define representations for many dataflow problems that summarize dataflow functions compactly.

In this section, we present our dataflow framework in three steps. The first framework covers the language without break statements; the second adds break statements; and the third — the full framework — adds the feature of assigning a dataflow value to each node rather than just to the root. For each of these three frameworks, the plan is the same:

1. Present an analysis framework $\mathcal{F}$ for calculating dataflow values for AST's in a lattice $Data$.
2. Present a framework $\mathcal{R}$ for calculating representations of dataflow functions, given an "adequate" representation R.
3. Give a theorem relating representations produced by $\mathcal{R}$ to dataflow functions given by $\mathcal{F}$.
4. Give an alternative method of calculating representations, called $\mathcal{F}^R$, more efficient than $\mathcal{R}$, which uses the definition of $\mathcal{F}$ but applies it to representations rather than dataflow values.

As a running example in these sections, we use *uninitialized variables*, an analysis that calculates a list of variables that may have been used without being initialized.

The first framework contains only simple control structures; the theorems are trivial in this case, but we introduce notation and explain how staging works. The second framework handles break statements. These two frameworks calculate dataflow values only for the root of an AST; the final framework calculates values at each node within an AST.

Figure 4.2 shows the abstract syntax of the language we treat in this chapter. We use a Java-like language for concrete syntax. Keep in mind that this is the language *inside quotations*. We do not include holes because these are not proper elements of the language. To avoid notational complexities, we allow holes only in statement position; allowing holes in expression position poses no fundamental problems.

$$\mathcal{F}[\![\mathsf{skip}]\!] = \mathit{id}$$
$$\mathcal{F}[\![x = e]\!] = \mathit{asgn}(x, e)$$
$$\mathcal{F}[\![P_1; P_2]\!] = \mathcal{F}[\![P_1]\!]; \mathcal{F}[\![P_2]\!]$$
$$\mathcal{F}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \mathit{exp}(e); (\mathcal{F}[\![P_1]\!] \wedge \mathcal{F}[\![P_2]\!])$$

Figure 4.3: First framework for data-flow analysis.

Dataflow values are assumed to come from a lattice, called $Data$. Define *DFFun* to be the function space $Data \to Data$ (confined to functions that preserve $\top_{Data}$).

4.1.1 Simple Control Structures

Our first framework (Figure 4.3) treats a subset of the full language, programs with only sequencing and conditionals. $\mathcal{F}$ assigns an element of *DFFun* to every program. We use semi-colon (;) for function composition in diagrammatic order. The meet ($\wedge$) operation on functions is defined pointwise, and *id* is the identity function in *DFFun*. Only the functions *asgn* and *exp* are specific to a particular analysis. The types of all the names appearing in this definition are:

$$\begin{aligned}
&\mathit{id} : \mathit{DFFun}\\
&\mathit{asgn} : \mathrm{Var} \times \mathrm{Exp} \to \mathit{DFFun}\\
&\mathit{exp} : \mathrm{Exp} \to \mathit{DFFun}\\
&; : \mathit{DFFun} \times \mathit{DFFun} \to \mathit{DFFun}\\
&\wedge : \mathit{DFFun} \times \mathit{DFFun} \to \mathit{DFFun}
\end{aligned}$$

We earlier stated that we allow only $\top$-preserving functions in *DFFun*. The identity function has this property, and function composition and meet preserve it, so we need only to confirm it for *asgn* and *exp* for each analysis.

To get the result of the static analysis of P, apply $\mathcal{F}[\![P]\!]$ to an appropriate initial value.

As an example, we define an analysis for variable initialization. Here, $Data = \mathcal{P}(Var)^2$, with ordering

$$(D, U) \sqsubseteq (D', U') \text{ if } D \subseteq D' \text{ and } U \supseteq U'$$

The datum (D, U) at a node means that D is the set of variables that definitely have definitions at this point, and U is the set that may have been used without definition.

$$\begin{aligned}
\mathit{asgn}(x, e) &= \lambda(D, U).(D \cup \{x\}, (vars(e) \setminus D) \cup U)\\
\mathit{exp}(e) &= \lambda(D, U).(D, (vars(e) \setminus D) \cup U)
\end{aligned}$$

$vars(e)$ is the set of variables occurring in e. It is easy to see that $\mathit{asgn}(x, e)$ and $\mathit{exp}(e)$ preserve $\top_{Data}$ (the pair $(Var, \varnothing)$).

$$\begin{aligned}
&\mathcal{R}[\![\mathsf{skip}]\!] = id_R \\
&\mathcal{R}[\![x = e]\!] = asgn_R(x, e) \\
&\mathcal{R}[\![P_1; P_2]\!] = \mathcal{R}[\![P_1]\!] \;;_R\; \mathcal{R}[\![P_2]\!] \\
&\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = exp_R(e) \;;_R\; (\mathcal{R}[\![P_1]\!] \wedge_R \mathcal{R}[\![P_2]\!])
\end{aligned}$$

Figure 4.4: Representation function for the first framework.

Returning to the general case, our task is to find representations of elements of *DFFun* for each analysis.

Definition 4.1.1. Suppose R is a set with the following values and functions ($\top_R$ is not used until the next subsection):

$$\begin{array}{llll}
\top_R & : R & exp_R\!: & \text{Exp} \to R \\
id_R & : R & ;_R & : R \times R \to R \\
asgn_R\!: & \text{Var} \times \text{Exp} \to R & \wedge_R & : R \times R \to R
\end{array}$$

R is an *adequate representation* of a dataflow problem if there is a homomorphism

$$\mathbf{abs} : R \to \mathit{DFFun}$$

from $(R, \top_R, id_R, asgn_R, exp_R, ;_R, \wedge_R)$ to $(\mathit{DFFun}, \top_{DFFun}, id, asgn, exp, ;, \wedge)$. Specifically, this requires

$$\begin{aligned}
\mathbf{abs}(\top_R) &= \top_{DFFun} = \lambda d.\top_{Data} \\
\mathbf{abs}(id_R) &= id \\
\mathbf{abs}(asgn_R(x, e)) &= asgn(x, e) \\
\mathbf{abs}(exp_R(e)) &= exp(e) \\
\mathbf{abs}(r \;;_R r') &= \mathbf{abs}(r); \mathbf{abs}(r') \\
\mathbf{abs}(r \wedge_R r') &= \mathbf{abs}(r) \wedge \mathbf{abs}(r')
\end{aligned}$$

Now, define $\mathcal{R} : \text{Pgm} \to R$ to be the function in Figure 4.4. Then we have the following theorem.

Theorem 4.1.2. *If R is an adequate representation, then for all P,* $\mathbf{abs}(\mathcal{R}[\![P]\!]) = \mathcal{F}[\![P]\!]$.

Proof. A trivial structural induction. □

For uninitialized variables, a natural representation, which is also adequate, is almost the same as $Data$:

$$R = \mathcal{P}(Var)^2 \cup \{\top_R\}$$

For any fragment P, $\mathcal{R}[\![P]\!]$ is the pair containing the set of variables definitely defined in P and the set possibly used without definition in P. The operations on this representation are as below.

$$\begin{aligned}
&id_R = (\varnothing, \varnothing)\\
&asgn_R(x, e) = (\{x\}, vars(e))\\
&exp_R(e) = (\varnothing, vars(e))\\
&(D, U) ;_R (D', U') = (D \cup D', U \cup (U' \setminus D))\\
&(D, U) \wedge_R (D', U') = (D \cap D', U \cup U')
\end{aligned}$$

Throughout the chapter, to avoid clutter, we ignore $\top$ when defining functions; in every case, the definitions of $asgn(x, e)$, $exp(e)$ should check for $\top_{Data}$, and $;_R$ and $\wedge_R$ should check for $\top_R$.

The **abs** function is defined as

$$\mathbf{abs}(D, U) = \lambda(D', U').(D' \cup D, U' \cup (U \setminus D'))$$

We note that **abs**($\top_R$) necessarily equals $\lambda d.\top_{Data}$, as required by the definition of adequacy.

To illustrate the analysis, we show a program annotated with the value of $\mathcal{R}[\![P]\!]$ for each subtree P:

```
// ({x,y},{x,z})  (entire fragment)
y = x;                // ({y},{x})
if (z>10)             // ({x},{x,y,z})  ('if' statement)
{                     // ({x,w},{x,y})  ('true' branch)
  w = 15;             // ({w},∅)
  x = x + y + w;      // ({x},{x,y,w})
} else
  x = 0;              // ({x},∅)
```

In Figure 4.2, we included while statements in our language. They can be defined using a maximal fixpoint in the usual way:

$$\mathcal{F}[\![\mathsf{while}(e)\ \mathsf{do}\ P]\!] = \mathit{mfxp}(\lambda p.\mathit{exp}(e); (\mathcal{F}[\![P]\!]; p \wedge id))$$

If we were to include $\mathcal{R}[\![\mathsf{while}(e)\ \mathsf{do}\ P]\!]$ in Figure 4.4, we would define it as $while_R(\mathcal{R}[\![e]\!], \mathcal{R}[\![P]\!])$, where $while_R$ is a function specific to each analysis. We will not mention this further, however, because in each of our examples, the function $while_R$ is not very interesting: $while_R(r_1, r_2)$ is either $r_1;_R r_2;_R r_1$ or $r_1;_R r_2;_R r_1;_R r_2;_R r_1$. That is, only a fixed number of iterations of the loop body is required.

In principle, we could now move on to staging, using $\mathcal{R}$ to calculate the representation of fragments. In practice, we calculate them by using the definition of $\mathcal{F}$. This method will turn out, when applied to the full framework, to be more efficient; see page 58. The difference is that $\mathcal{R}$ is a purely bottom-up algorithm (producing and composing functions),

$$\begin{aligned}
&\mathcal{F}^R[\![\mathsf{skip}]\!] = id^R \\
&\mathcal{F}^R[\![x = e]\!] = asgn^R(x, e) \\
&\mathcal{F}^R[\![P_1; P_2]\!] = \mathcal{F}^R[\![P_1]\!] \ ; \ \mathcal{F}^R[\![P_2]\!] \\
&\mathcal{F}^R[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = exp^R(e) \ ; \ (\mathcal{F}^R[\![P_1]\!] \wedge^R \mathcal{F}^R[\![P_2]\!])
\end{aligned}$$

Figure 4.5: $\mathcal{F}^R$ for the first framework.

while $\mathcal{F}$ is more top-down (threading an analysis result through transfer functions). The situation is similar to the use of an accumulator parameter in functional programs, which can turn a quadratic algorithm into a linear one [IB99].

Define $\mathcal{F}^R : \text{Pgm} \to R \to R$ to be the function in Figure 4.5, with the relevant operations defined as follows:

$$\begin{aligned}
&id^R = id \\
&asgn^R(x, e) = \lambda r.r \,;_R\, asgn_R(x, e) \\
&exp^R(e) = \lambda r.r \,;_R\, exp_R(x, e) \\
&f \wedge^R g = \lambda r.fr \wedge_R gr
\end{aligned}$$

Definition 4.1.3. Two representation values, r and r', are equivalent, denoted $r \equiv r'$, if $\mathbf{abs}(r) = \mathbf{abs}(r')$.

Theorem 4.1.4. *If R is adequate, then for all P and r, $\mathcal{F}^R[\![P]\!]r \equiv r \,;_R\, \mathcal{R}[\![P]\!]$.*

Proof. By induction on the structure of P. □

Corollary 4.1.5. $\mathcal{F}^R[\![P]\!]id_R \equiv \mathcal{R}[\![P]\!]$.

Proof.

$$\begin{aligned}
\mathbf{abs}(\mathcal{F}^R[\![P]\!]id_R) &= \mathbf{abs}(id_R;_R \mathcal{R}[\![P]\!]) \\
&= \mathbf{abs}(id_R); \mathbf{abs}(\mathcal{R}[\![P]\!]) \\
&= id; \mathbf{abs}(\mathcal{R}[\![P]\!]) \\
&= \mathbf{abs}(\mathcal{R}[\![P]\!])
\end{aligned}$$

□

If **abs** is injective — in which case we call R an *exact representation* — then we can replace $\equiv$ by $=$ in the above theorems. All the analyses we define in this chapter are exact.

We are now ready to stage static analyses, as depicted in Figure 4.1. The first stage calculates values of R, using $\mathcal{F}^R$, and the second, run-time, stage uses $\mathcal{F}$ to complete the analysis.

$$
\begin{aligned}
&\mathcal{F}[\![\mathsf{skip}]\!] = \mathit{id}\\
&\mathcal{F}[\![x = e]\!] = \lambda(\eta, d).(\eta, \mathit{asgn}(x, e)(d))\\
&\mathcal{F}[\![\mathsf{break}\ \ell]\!] = \lambda(\eta, d).(\eta[\ell \mapsto d \wedge \eta(\ell)], \top_{Data})\\
&\mathcal{F}[\![\ell : P]\!] = \lambda(\eta, d).\ \mathsf{let}\ (\eta_1, d_1) \leftarrow \mathcal{F}[\![P]\!](\eta, d)\\
&\qquad\qquad\qquad\qquad \mathsf{in}\ (\eta_1[\ell \mapsto \top_{Data}], d_1 \wedge \eta_1(\ell))\\
&\mathcal{F}[\![P_1; P_2]\!] = \mathcal{F}[\![P_1]\!]; \mathcal{F}[\![P_2]\!]\\
&\mathcal{F}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta, d).\ \mathsf{let}\ (\eta_1, d_1) \leftarrow \mathcal{F}[\![P_1]\!](\eta, \mathit{exp}(e)(d))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\eta_2, d_2) \leftarrow \mathcal{F}[\![P_2]\!](\eta, \mathit{exp}(e)(d))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ (\eta_1, d_1) \wedge (\eta_2, d_2)
\end{aligned}
$$

Figure 4.6: Framework with break statements.

4.1.2 Break Statements

We expand our analysis now to labelled statements and break-to-label statements. We will see that an adequate representation in the sense of the previous section can be extended uniformly to a representation for this case.

Throughout the chapter, we assume all programs are legal in the sense that they do not contain nested labelled statements with the same label.

An *environment* η is a function in $Env = \text{Label} \rightarrow Data$. Now the incoming and outgoing values are pairs:

$$\mathcal{F}[\![P]\!] : Env \times Data \rightarrow Env \times Data$$

The extended analysis is shown in Figure 4.6. *asgn* and *exp* have the same types as in the previous section; semi-colon is again function composition (in the expanded space), and *id* is the identity function. We extend meet to environments element-wise and then to pairs component-wise.

To explain Figure 4.6: Suppose a statement P is contained within a labelled statement with label ℓ, and we are evaluating $\mathcal{F}[\![P]\!](\eta, d)$. The argument d contains information about the control flow paths that reach P. The environment η contains information about all the control flow paths that were terminated with a break ℓ statement prior to reaching P; since there may be more than one, $\eta(\ell)$ gives a conservative approximation by taking the meet of all those paths. Thus, if P is break ℓ, then d is incorporated into the outgoing environment by taking $d \wedge \eta(\ell)$. Furthermore, the "normal exit" from P is $\top_{Data}$, which ensures that any statement directly following P will be ignored (since, for any statement Q, $\mathcal{F}[\![Q]\!]$ preserves $\top_{Data}$ in its second argument). Now consider labelled statements. $\mathcal{F}[\![\ell : P]\!](\eta, d)$ first calculates $\mathcal{F}[\![P]\!](\eta, d)$. A normal exit from $\ell : P$ can be a normal exit from P or a break to ℓ, so we take the meet of these two values. Furthermore, the binding of ℓ in the environment is reset to $\top_{Data}$, since a subsequent statement could be labelled ℓ.

$$
\begin{aligned}
&\mathcal{R}[\![\mathsf{skip}]\!] = (\top_{Env_R}, id_R)\\
&\mathcal{R}[\![x = e]\!] = (\top_{Env_R}, asgn_R(x, e))\\
&\mathcal{R}[\![\mathsf{break}\ \ell]\!] = (\top_{Env_R}[\ell \mapsto id_R], \top_R)\\
&\mathcal{R}[\![\ell : P]\!] = \text{let } (\eta, r) \leftarrow \mathcal{R}[\![P]\!]\\
&\qquad\qquad \text{in } (\eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))\\
&\mathcal{R}[\![P_1; P_2]\!] = \text{let } (\eta_1, r_1) \leftarrow \mathcal{R}[\![P_1]\!], (\eta_2, r_2) \leftarrow \mathcal{R}[\![P_2]\!]\\
&\qquad\qquad \text{in } (\eta_1 \wedge_R (r_1;_R \eta_2), r_1;_R r_2)\\
&\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \text{let } (\eta_1, r_1) \leftarrow \mathcal{R}[\![P_1]\!], (\eta_2, r_2) \leftarrow \mathcal{R}[\![P_2]\!]\\
&\qquad\qquad\qquad\qquad \text{in } \exp_R(e);_R ((\eta_1, r_1) \wedge_R (\eta_2, r_2))
\end{aligned}
$$

Figure 4.7: Representation for framework of Figure 4.6.

Representations of these functions are derived from representations of functions in *DFFun*. Assume R is an adequate representation of *DFFun*. It can be extended to a representation E_R of functions in the space $Env \times Data \to Env \times Data$. Define $Env_R =$ Label $\to R$. Then

$$E_R \;=\; Env_R \times R$$

Figure 4.7 gives a function to calculate representations. Although very similar to $\mathcal{F}$, $\mathcal{R}$ has one crucial difference: For statement $P_1; P_2$, where $\mathcal{F}$ simply uses function composition, $\mathcal{R}$ calculates an explicit value. Of particular interest is the way environments are affected. The environment given by $\mathcal{R}[\![P_2]\!]$ incorporates all the control flow up to any break statements in P_2. The new environment augments each value in that environment by adding r_1, which is the dataflow information for a *normal* exit from P_1. That is, an abnormal (to an enclosing label) exit is either an abnormal exit from P_1 or a normal exit from P_1 followed by an abnormal exit from P_2. Furthermore, if there is a break to the same label from both P_1 and P_2, the total effect is that two separate paths meet after the statement with that label, so the functions in the two environments are joined.

Defining the abstraction function:

$$
\begin{aligned}
&\mathbf{abs}_E : E_R \to (Env \times Data \to Env \times Data)\\
&\mathbf{abs}_E(\eta_R, r) \;=\; \lambda(\eta, d).(\lambda\ell.\eta(\ell) \wedge \mathbf{abs}(\eta_R(\ell))d, \mathbf{abs}(r)d)
\end{aligned}
$$

we have the following theorem.

Theorem 4.1.6. *If R is adequate, then for any legal program P, $\mathbf{abs}_E(\mathcal{R}[\![P]\!]) = \mathcal{F}[\![P]\!]$.*

Proof. By induction on the structure of P. □

Again, we can (and do) calculate $\mathcal{R}$ by reinterpreting $\mathcal{F}$ using the operators of R. The function

$$\mathcal{F}^R : \text{Pgm} \to E_R \to E_R$$

$$\begin{aligned}
&\mathcal{F}^R[\![\mathsf{skip}]\!] = id^R \\
&\mathcal{F}^R[\![x = e]\!] = \lambda(\eta, r).(\eta, asgn^R(x, e)r) \\
&\mathcal{F}^R[\![\mathsf{break}\ \ell]\!] = \lambda(\eta, r).(\eta[\ell \mapsto r \wedge_R \eta(\ell)], \top_R) \\
&\mathcal{F}^R[\![\ell : P]\!] = \lambda(\eta, r).\ \mathsf{let}\ (\eta_1, r_1) \leftarrow \mathcal{F}^R[\![P]\!](\eta, r) \\
&\qquad\qquad\qquad\qquad\quad \mathsf{in}\ (\eta_1[\ell \mapsto \top_R], r_1 \wedge_R \eta_1(\ell)) \\
&\mathcal{F}^R[\![P_1; P_2]\!] = \mathcal{F}^R[\![P_1]\!]; \mathcal{F}^R[\![P_2]\!] \\
&\mathcal{F}^R[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta, r).\ \mathsf{let}\ (\eta_1, r_1) \leftarrow \mathcal{F}^R[\![P_1]\!](\eta, exp^R(e)r) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\eta_2, r_2) \leftarrow \mathcal{F}^R[\![P_2]\!](\eta, exp^R(e)r) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ (\eta_1, r_1) \wedge_R (\eta_2, r_2)
\end{aligned}$$

Figure 4.8: $\mathcal{F}^R$ with break statements.

is defined as given in Figure 4.8 where $asgn^R$ and exp^R are exactly the same as in the previous section; id^R has the same definition but different type.

Theorem 4.1.7. *Let P be a legal program, and $(\eta, r) = \mathcal{R}[\![P]\!]$. Then, for all η' and r', as long as $\eta'(L) = \top_R$ for any label L that occurs in P, we have*

$$\mathcal{F}^R[\![P]\!](\eta', r') \equiv (\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')), r' ;_R r).$$

Proof. By induction on the structure of P. □

Corollary 4.1.8. $\mathcal{F}^R[\![P]\!](\top_{Env_R}, id_R) \equiv \mathcal{R}[\![P]\!]$.

Proof. Let $\mathcal{R}[\![P]\!] = (\eta, r)$. Then, by the theorem above,

$$\begin{aligned}
\mathcal{F}^R[\![P]\!](\top_{Env_R}, id_R) &\equiv (\lambda\ell'.\top_{Env_R}(\ell') \wedge_R (id_R ;_R \eta(\ell')), id_R ;_R r) \\
&= (\lambda\ell'.\top_R \wedge_R (id_R ;_R \eta(\ell')), id_R ;_R r)
\end{aligned}$$

which means

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{F}^R[\![P]\!](\top_{Env_R}, id_R)) = \mathbf{abs}_E((\lambda\ell'.\top_R \wedge_R (id_R ;_R \eta(\ell')),\ id_R ;_R r)) \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\top_R \wedge_R (id_R ;_R \eta(\ell')))d'', \mathbf{abs}(id_R ;_R r)d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\top_R)d'' \wedge (\mathbf{abs}(id_R); \mathbf{abs}(\eta(\ell')))d'', (\mathbf{abs}(id_R); \mathbf{abs}(r))d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta(\ell'))d'', \mathbf{abs}(r)d'') \\
&\quad = \mathbf{abs}_E((\eta, r)) \\
&\quad = \mathbf{abs}_E(\mathcal{R}[\![P]\!])
\end{aligned}$$

□

Again, $\equiv$ can be replaced by $=$ for all the analyses we present in this chapter.

Adding a break statement to our previous example on page 52, we show the values of $\mathcal{F}^R[\![P]\!](\top_{Env_R}, (\varnothing, \varnothing))$ for each node P.

```
                        // ({L↦({x,y},{x,z})},({x,w,y},{x,z}))
y = x;                  // (∅, ({y},{x}))
if (z>10)               // ({L↦({x},{z})}, ({x,w},{x,y,z}))
{                       // (∅, ({x,w},{x,y}))
  w = 15;               // (∅, ({w},∅))
  x = x + y + w;        // (∅, ({x},{x,y,w}))
} else
{                       // ({L↦({x},∅)},⊤)
  x = 0;                // (∅, ({x},∅))
  break L;              // ({L↦(∅,∅)},⊤)
}
```

Note that in the topmost node, w is in the defined set for normal exit even though it is not defined in both branches of the if-statement. This is because the flow reaches the end of the if-statement only if the then-branch where w is defined is taken.

The approach to staging is unchanged.

4.1.3 The Framework

The frameworks described so far lack one important ingredient: they do not give us information about each node in the AST, but only about the root node of the AST. Most static analyses are used to obtain information at each node: What definitions reach this particular node? What variables have constant values at this particular point in the program? Etc.

The complete analysis returns a map giving data at each node. Assuming each node in a Pgm is uniquely identified by an element of $Node$, we define $NodeMap = Node \rightharpoonup \mathit{Data}$ (partial functions from $Node$ to Data). Now,

$$\mathcal{F}[\![P]\!] : NodeMap \times Env \times \mathit{Data} \to NodeMap \times Env \times \mathit{Data}$$

We also change the type of *asgn*:

$$\mathit{asgn} : Node \times \mathrm{Var} \times \mathrm{Exp} \to \mathit{DFFun}$$

for cases (such as reaching definitions) where $Node$ is contained within Data. In cases such as uninitialized variables, the first argument is ignored. The full forward analysis is shown in Figure 4.9.

$$
\begin{aligned}
&\mathcal{F}[\![n : \mathsf{skip}]\!] = \lambda(\varphi, \eta, d).(\varphi[n \mapsto d], \eta, d)\\
&\mathcal{F}[\![n : x = e]\!] = \lambda(\varphi, \eta, d).\mathsf{let}\ d' \leftarrow \mathit{asgn}(n, x, e)(d)\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ (\varphi[n \mapsto d'], \eta, d')\\
&\mathcal{F}[\![n : \mathsf{break}\ \ell]\!] = \lambda(\varphi, \eta, d).(\varphi[n \mapsto \top_{Data}], \eta[\ell \mapsto d \wedge \eta(\ell)], \top_{Data})\\
&\mathcal{F}[\![n : (\ell : (n_1 : P))]\!] = \lambda(\varphi, \eta, d).\ \mathsf{let}\ (\varphi_1, \eta_1, d_1) \leftarrow \mathcal{F}[\![n_1 : P]\!](\varphi, \eta, d)\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ (\varphi_1[n \mapsto d_1 \wedge \eta_1(\ell)], \eta_1[\ell \mapsto \top_{Data}], d_1 \wedge \eta_1(\ell))\\
&\mathcal{F}[\![n : (n_1 : P_1; \quad n_2 : P_2)]\!] = \lambda(\varphi, \eta, d).\ \mathsf{let}\ (\varphi_1, \eta_1, d_1) \leftarrow \mathcal{F}[\![n_2 : P_2]\!](\mathcal{F}[\![n_1 : P_1]\!](\varphi, \eta, d))\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ (\varphi_1[n \mapsto d_1], \eta_1, d_1)\\
&\mathcal{F}[\![n : \mathsf{if}(e)\ n_1 : P_1\ \mathsf{else}\ n_2 : P_2]\!] = \lambda(\varphi, \eta, d).\ \mathsf{let}\ (\varphi_1, \eta_1, d_1) \leftarrow \mathcal{F}[\![n_1 : P_1]\!](\varphi, \eta, \mathit{exp}(e)(d))\\
&\qquad\qquad\qquad\qquad\qquad (\varphi_2, \eta_2, d_2) \leftarrow \mathcal{F}[\![n_2 : P_2]\!](\varphi, \eta, \mathit{exp}(e)(d))\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ ((\varphi_1 \cup \varphi_2)[n \mapsto d_1 \wedge d_2], \eta_1 \wedge \eta_2, d_1 \wedge d_2)
\end{aligned}
$$

Figure 4.9: Forward analysis framework.

As in the previous section, we can start with an adequate representation and create a representation for this analysis. Specifically, define

$$F_R = (Node \rightharpoonup R) \times Env_R \times R$$

The abstraction function becomes:

$$
\begin{aligned}
&\mathbf{abs}_F : F_R \to (NodeMap \times Env \times Data \to NodeMap \times Env \times Data)\\
&\mathbf{abs}_F(\varphi_R, \eta_R, r) \ = \ \lambda(\varphi', \eta', d').(\varphi' \cup (\lambda n.\mathbf{abs}(\varphi_R(n))d'),\ \lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\eta_R(\ell))d',\ \mathbf{abs}(r)d')
\end{aligned}
$$

Representations are calculated by function $\mathcal{R}$ as given in Figure 4.10.

Theorem 4.1.9. *If R is adequate, then for any legal program P, $\mathbf{abs}_F(\mathcal{R}[\![P]\!]) = \mathcal{F}[\![P]\!]$.*

Proof. Similar to the proof for the intermediate framework (Theorem 4.1.6). □

We can define $\mathcal{F}^R$ as in previous sections, and obtain

Theorem 4.1.10. *Let P be a legal program and $(\varphi, \eta, r) = \mathcal{R}[\![P]\!]$. Then for all φ', η' and r', as long as $\eta'(L) = \top_R$ for any label L that occurs in P, we have*

$$\mathcal{F}^R[\![P]\!](\varphi', \eta', r') \equiv (\varphi' \cup \lambda n.r' ;_R \varphi(n), \lambda l.\eta(l) \wedge_R (r' ;_R \eta(l)), r' ;_R r)$$

Proof. Similar to the proof for the intermediate framework (Theorem 4.1.7). □

The importance of $\mathcal{F}^R$ can now be explained. $\mathcal{R}$ calculates the node map φ bottom-up. Suppose $\mathcal{R}[\![P]\!] = (\varphi, \eta, r)$, and consider $\varphi(n)$, where n is a node in P. $\varphi(n)$ says how to calculate a data value at n given input data at P; that is, it represents the computation from the start of P to n. In calculating $\mathcal{R}[\![P_1; P_2]\!]$, the subcomputation $\mathcal{R}[\![P_2]\!]$ returns a

$$
\begin{aligned}
&\mathcal{R}[\![n : \mathsf{skip}]\!] = (\{n \mapsto id_R\}, \top_{Env_R}, id_R)\\
&\mathcal{R}[\![n : x = e]\!] = (\{n \mapsto asgn_R(n,x,e)\}, \top_{Env_R}, asgn_R(n,x,e))\\
&\mathcal{R}[\![n : \mathsf{break}\ \ell]\!] = (\{n \mapsto \top_R\}, \top_{Env_R}[\ell \mapsto id_R], \top_R)\\
&\mathcal{R}[\![n : (\ell : (n_1 : P))]\!] = \text{let } (\varphi, \eta, r) \leftarrow \mathcal{R}[\![n_1 : P]\!]\\
&\qquad\qquad\qquad\qquad \text{in } (\varphi[n \mapsto r \wedge_R \eta(\ell)], \eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))\\
&\mathcal{R}[\![n : (n_1 : P_1; \quad n_2 : P_2)]\!] = \text{let } (\varphi_1, \eta_1, r_1) \leftarrow \mathcal{R}[\![n_1 : P_1]\!], (\varphi_2, \eta_2, r_2) \leftarrow \mathcal{R}[\![n_2 : P_2]\!]\\
&\qquad\qquad\qquad\qquad\qquad \text{in } (\lambda n'.\ \text{if } \varphi_1(n') \text{ defined then } \varphi_1(n')\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } \varphi_2(n') \text{ defined then } r_1;_R\, \varphi_2(n')\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } n' = n \text{ then } r_1;_R\, r_2\ ,\\
&\qquad\qquad\qquad\qquad\qquad\quad \eta_1 \wedge_R (r_1;_R\, \eta_2),\\
&\qquad\qquad\qquad\qquad\qquad\quad r_1;_R\, r_2)\\
&\mathcal{R}[\![n : \mathsf{if}(e)\ n_1 : P_1\ \mathsf{else}\ n_2 : P_2]\!] = \text{let } (\varphi_1, \eta_1, r_1) \leftarrow \mathcal{R}[\![n_1 : P_1]\!], (\varphi_2, \eta_2, r_2) \leftarrow \mathcal{R}[\![n_2 : P_2]\!]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{in } (exp_R(e);_R\, ((\varphi_1 \cup \varphi_2)[n \mapsto (r_1 \wedge_R r_2)]),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad exp_R(e);_R\, (\eta_1 \wedge_R \eta_2),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad exp_R(e);_R\, (r_1 \wedge_R r_2))
\end{aligned}
$$

Figure 4.10: Representation for framework of Figure 4.9.

node map φ_2 representing computations *within* P_2. The result for $P_1; P_2$ has to give values for nodes *in* P_2 that represent the computation *starting at* P_1. Thus, it not only produces values for each node in P_1, but also calculates *new* values for every node in P_2. Extending this reasoning to a list of statements $P_1; \ldots; P_n$, we see that values for all the nodes in P_n will be calculated n times, for all the nodes in P_{n-1} $n-1$ times, etc. Thus, the complexity of $\mathcal{R}[\![P]\!]$ is quadratic in the size of the P. $\mathcal{F}$ uses, in effect, an accumulator, passing φ through the entire tree, and thus calculates a value for each node just once.

Our previous example (on page 57) with numbered nodes is in Figure 4.11. We show the value of $\mathcal{R}[\![P]\!]$ only at the top node. The environment and data values are just as in Section 4.1.2: {L↦({x,y},{x,z})} and ({x,w,y},{x,z}), respectively. The node map is:

$$
\begin{aligned}
\{\ & n_1 \mapsto (\{x,w,y\},\{x,z\}),\\
& n_2 \mapsto (\{y\},\{x\}),\\
& n_3 \mapsto (\{x,w,y\},\{x,z\}),\\
& n_4 \mapsto (\{x,w,y\},\{x,z\}),\\
& n_5 \mapsto \top_R,\\
& n_6 \mapsto (\{w,y\},\{x,z\}),\\
& n_7 \mapsto (\{x,w,y\},\{x,z\}),\\
& n_8 \mapsto (\{x,y\},\{x,z\}),\\
& n_9 \mapsto \top_R\\
\}\ &
\end{aligned}
$$

Note that the values associated with the nodes are different from those in the previous analyses. This node map incorporates what is known about each node *at the top node* (as

```
n1:  // entire fragment
n2:  y = x;
n3:  if (z>10)
n4:  {
n6:    w = 15;
n7:    x = x + y + w;
     } else
n5:  {
n8:    x = 0;
n9:    break L;
     }
```

Figure 4.11: The example program with numbered nodes.

in [SP81]). For example, when we get through node n_6, we will have defined w and y, and will have used x and z possibly without definition. Thus, suppose we put this fragment into a hole at a position where x has been defined. We can look at, for example, node n_6 and immediately find that only z may have been used without definition. In general, we have the chance to query the data of selected nodes without analyzing the entire tree, which can have a salutary effect on the run-time performance of the analysis. Note also that the fragment as a whole definitely defines w, even though it is only defined in one branch of the conditional; since the else-branch ends in a break, control can only reach the end of this statement by taking the then-branch.

Again, staging is not fundamentally different in this more complicated framework. One new wrinkle is that a single plug cannot be used to fill in two holes because its node names would then not be unique in the larger AST; thus, nodes in plugs need to be uniformly renamed before insertion in a larger tree, a process that is easily done.

4.2 Adequate Representations

We now present several analyses. Like variable initialization, all the representations we present here are exact.

4.2.1 Reaching Definitions I (RD)

The reaching definitions (RD) at a point in a program include any assignment statement which may have been the most recent assignment to a variable prior to this point. Representations for this analysis have been given in [MR90, RHS95, RKM06].

$$D \in Data = \mathcal{P}(\text{Node}) \cup \{\top\}$$

Sets in $Data$ are ordered by reverse inclusion, with $\varnothing$ being the element just below $\top$. The operations are

$$\begin{aligned}&\mathit{asgn}(n, x, e) = \lambda D.(D \setminus D_x) \cup \{n\}\\ &\mathit{exp}(e) = \lambda D.D\end{aligned}$$

where D_x are the nodes that define x. The representation is:

$$R = (\mathcal{P}(\mathrm{Var}) \times \mathcal{P}(\mathrm{Node})) \cup \{\top_R\}$$

Suppose $K \in \mathcal{P}(\mathrm{Var})$ and $G \in \mathcal{P}(\mathrm{Node})$. If $\mathcal{R}[\![P]\!] = (K, G)$, the set K contains all the variables definitely defined in P, and G are the assignment statements that define those variables and may reach the end of P.

$$\begin{aligned}&\mathit{id}_R = (\varnothing, \varnothing)\\ &\mathit{asgn}_R(n, x, e) = (\{x\}, \{n\})\\ &\mathit{exp}_R(e) = (\varnothing, \varnothing)\\ &(K_1, G_1) ;_R (K_2, G_2) = (K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2))\\ &(K_1, G_1) \wedge_R (K_2, G_2) = (K_1 \cap K_2, G_1 \cup G_2)\end{aligned}$$

$$\mathbf{abs}(K, G) = \lambda D.\, G \cup (D \setminus K)$$

where $G \setminus K = \{n \in G \mid n \text{ is not the definition of some } x \in K\}$.

Theorem 4.2.1. *R for* RD *is an exact representation.*

Proof. Given in the appendix. □

4.2.2 Available Expressions (AE)

Available expressions (AE) are those expressions that have been previously computed, such that no intervening assignment has made their value obsolete. A given statement makes some expressions available, kills some expressions (by assigning to the variables they contain), and lets others pass through unmolested.

$$E \in Data = \mathcal{P}(\mathrm{Exp}) \cup \{\top\}$$

Sets in $Data$ are ordered by set inclusion.

$$\begin{aligned}&\mathit{asgn}(n, x, e) = \lambda E.(E \cup sub(e)) \setminus E_x\\ &\mathit{exp}(e) = \lambda E.E \cup sub(e)\end{aligned}$$

where E_x is the set of expressions in E that contain x, and $sub(e)$ is the set of all subexpressions of e.

The following seems an obvious representation.

$$R = (\mathcal{P}(\mathrm{Var}) \times \mathcal{P}(\mathrm{Exp})) \cup \{\top_R\}$$

The R value (K, G) represents that G is the set of expressions made available by a statement, and K is the set of variables defined by that statement (so that the statement kills any expressions containing those variables).

$$\begin{aligned}
&\mathit{id}_R = (\varnothing, \varnothing)\\
&\mathit{asgn}_R(n, x, e) = (\{x\}, \{e' \,|\, e' \in \mathit{sub}(e), x \notin \mathit{vars}(e')\})\\
&\mathit{exp}_R(e) = (\varnothing, \{e' \,|\, e' \in \mathit{sub}(e)\})\\
&(K_1, G_1);_R (K_2, G_2) = (K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2))\\
&(K_1, G_1) \wedge_R (K_2, G_2) = (K_1 \cup K_2, G_1 \cap G_2)
\end{aligned}$$

$$\mathbf{abs}(K, G) = \lambda E. G \cup (E \setminus K)$$

where $G \setminus K = \{e \in G \,|\, \text{none of the variables in } e \text{ occur in } K\}$.

However, this is not an adequate representation for the analysis. Consider the statement: if (cond) $\{a = \ldots;\ \ldots = a + b\}$ else $\{\}$. Suppose that $a + b$ is available before this statement. It will also be available afterwards. However, since there is an assignment to a in one branch, the statement kills any expression containing a. Furthermore, $a + b$ is not generated in the other branch. Thus, the representation of this if-statement is $(\{a\}, \varnothing)$. But this will kill the incoming definition of $a + b$.

To obtain an adequate representation, we need to record that some expressions are guaranteed to survive a statement, even if they contain variables that are in its kill set, while others will be killed, as usual. We do this by putting annotations on expressions in the available set:

Definition 4.2.2. For set S, $S_{\mathit{Annot}} = \{s_{\mathit{must}} \,|\, s \in S\} \cup \{s_{\mathit{sur}} \,|\, s \in S\}$. Also define the operation "$\cdot$" on annotations: $\mathit{must} \cdot \mathit{must} = \mathit{must}$ and otherwise $\alpha \cdot \alpha' = \mathit{sur}$, where α, α' are annotations.

Our analysis uses the set $\mathrm{Exp}_{\mathit{Annot}}$. The annotation sur stands for the case when there is some path in the fragment that lets the incoming expression survive. The annotation must stands for the case when there is no such path, so that the statement itself *must* define the expression if it is to be available. The dot operation encapsulates the notion that an expression can survive a conditional statement as long as it can survive at least one of the branches. Then, this analysis is defined as follows:

$$R = \mathcal{P}(\mathrm{Var}) \times \mathcal{P}(\mathrm{Exp}_{Annot}) \cup \{\top_R\}$$

$$\begin{aligned}
&id_R = (\varnothing, \varnothing)\\
&asgn_R(n, x, e) = (\{x\}, \{e'_{must} \,|\, e' \in sub(e), x \notin vars(e')\}\\
&exp_R(e) = (\varnothing, \{e'_{must} \,|\, e' \in sub(e)\})
\end{aligned}$$

$$\begin{aligned}
&(K_1, G_1);_R (K_2, G_2) = (K_1 \cup K_2,\\
&\quad \{e_{must} \,|\, e_{must} \in G_2\} \cup\\
&\quad \{e_\alpha \,|\, e_{sur} \in G_2,\ e_\alpha \in G_1\} \cup\\
&\quad \{e_{sur} \,|\, e_{sur} \in G_2,\ e_\alpha \notin G_1,\ vars(e) \cap K_1 = \varnothing\} \cup\\
&\quad \{e_\alpha \,|\, e_\alpha \in G_1,\ e'_\alpha \notin G_2,\ vars(e) \cap K_2 = \varnothing\})\\
&(K_1, G_1) \wedge_R (K_2, G_2) = (K_1 \cup K_2,\\
&\quad \{e_{\alpha \cdot \alpha'} \,|\, e_\alpha \in G_1,\ e'_\alpha \in G_2\} \cup\\
&\quad \{e_{sur} \,|\, e_\alpha \in G_1,\ e'_\alpha \notin G_2,\ vars(e) \cap K_2 = \varnothing\} \cup\\
&\quad \{e_{sur} \,|\, e_\alpha \in G_2,\ e'_\alpha \notin G_1,\ vars(e) \cap K_1 = \varnothing\})
\end{aligned}$$

$$\begin{aligned}
&\mathbf{abs}(K, G) = \lambda E.\{e \,|\, e_{must} \in G\} \cup\\
&\quad \{e \,|\, e_{sur} \in G,\ e \in E\} \cup\\
&\quad \{e \,|\, e \in E,\ e_\alpha \notin G,\ vars(e) \cap K = \varnothing\}
\end{aligned}$$

The most interesting case is in the definition of semicolon, when $e_{sur} \in G_2$ and $e \in G_1$ (with either annotation). In that case, e is included in the available set, *even if it is killed by* K_2. Looking again at the if statement we discussed above, the true branch gives $(\{a\}, \{(a+b)_{must}\})$, and the false branch gives $(\varnothing, \varnothing)$. The meet of these values is $(\{a\}, \{(a+b)_{sur}\})$. This value summarizes the effect of the if statement correctly: if $(a+b)$ is in the incoming available set, then it will be in the resulting available set.

Theorem 4.2.3. *R for* AE *is an exact representation.*

Proof. Similar to the proof for RD in Section 4.2.1. □

4.2.3 Reaching Definitions II (RD2)

Using annotations, we give an alternative representation for reaching definitions. We will call this analysis RD2. Here we annotate sets of definitions of a variable; a *must* subscript indicates that the set includes all possible definitions of the variable, while a *sur* subscript indicates that there is some path in this statement through which a previous definition of the variable might survive.

Let $N \in \mathcal{P}(Node)$ in the following definitions.

$$S \in R = (\text{Var} \rightarrow \mathcal{P}(Node)_{Annot}) \cup \{\top_R\}$$

$$\begin{aligned}
&id_R = \lambda v.\varnothing_{sur} \\
&asgn(n, x, e) = (\lambda v.\varnothing_{sur})[x \mapsto \{n\}_{must}] \\
&exp(e) = \lambda v.\varnothing_{sur}
\end{aligned}$$

$$\begin{aligned}
S_1;_R S_2 = \lambda x.\,& \text{let } N_\alpha \leftarrow S_1(x), N'_{\alpha'} \leftarrow S_2(x) \\
& \text{in if } \alpha' = must \text{ then } N'_{\alpha'} \text{ else } (N \cup N')_\alpha \\
S_1 \wedge_R S_2 = \lambda x.&\text{let } N_\alpha \leftarrow S_1(x), N'_{\alpha'} \leftarrow S_2(x) \\
& \text{in } (N \cup N')_{\alpha \cdot \alpha'}
\end{aligned}$$

We assume that $S(x)$ defaults to $\varnothing_{sur}$. Finally, the abstraction function is

$$\begin{aligned}
\textbf{abs}(S) \;\; = \lambda D. \;\; & \{n \in D \mid n : x = e \text{ and } S(x) = N_{sur}\} \cup \\
& \{n \in N \mid n : x = e \text{ and } S(x) = N_\alpha\}
\end{aligned}$$

where $D \in Data = \mathcal{P}(Node) \cup \{\top\}$ as before.

Theorem 4.2.4. *R for* RD2 *is an exact representation.*

Proof. Similar to the proof for RD in Section 4.2.1. □

4.2.4 Constant Propagation (CP)

The framework can be instantiated for constant propagation (CP) with the following definitions. For simplicity we consider only integers as constant values, and assume that the expressions in the language are arithmetic operations. A graph-based representation for this analysis can be found in [RHS95, SRH96]. That representation requires that the set of program variables be available to construct the representation graphs. By the nature of our context, we cannot, and do not, make such an assumption.

$$M \in Data = (\text{Var} \rightarrow \mathbb{Z}_\perp^\top) \cup \{\top_R\}$$

Function values in $Data$ are ordered under the usual pointwise ordering.

$$\begin{aligned}
&asgn(n, x, e) = \lambda M.\, \text{if } isConstant(e, M) \text{ then } M[x \mapsto consVal(e, M)] \\
&\qquad\qquad\qquad\quad \text{else } M[x \mapsto \perp] \\
&exp(e) = \lambda M.M
\end{aligned}$$

where *isConstant*(e, M) returns true if the expression e can be shown to have a constant value based on the values kept in the constant map M, and *consVal*(e, M) returns that constant value[1].

For the representation, R is a function giving values for variables. However, these values are actually sets of variables, integer literals, and binary expressions, meaning "the set will be reduced to a constant c, if every element it contains eventually reduces to the constant c". Using this set, we effectively delay the meet operation, and gradually complete it as information becomes available.

$$R = Var \to CS_{Annot}$$
$$CS = P(Exp \cup \{\bot\})$$

Implicitly, a $C \in CS$ is normalized to $\{\bot\}$ if it contains $\bot$ or two distinct integers.

As in the previous cases, the annotations are used to preserve information in conditionals. A *must* annotation on a set of expressions indicates that the variable they define is definitely assigned one of those expressions; a *sur* annotation indicates that some other definition may apply to that variable (but may, of course, assign the same value to it that these expressions do).

$$\begin{aligned}
&\mathit{id}_R = \lambda v.\varnothing_{sur}\\
&\mathit{asgn}_R(n, x, e) = (\lambda v.\varnothing_{sur})[x \mapsto \{e\}_{must}]\\
&\mathit{exp}_R(e) = \lambda v.\varnothing_{sur}
\end{aligned}$$

$$\begin{aligned}
M_1 \wedge_R M_2 &= \lambda x. M_1(x) \wedge_R M_2(x)\\
&= \lambda x. \text{let } C_\alpha \leftarrow M_1(x), C'_{\alpha'} \leftarrow M_2(x)\\
&\qquad \text{in } (C \cup C')_{\alpha\cdot\alpha'}
\end{aligned}$$

$$\begin{aligned}
&M_1;_R M_2 = \lambda x.\mathit{semicolon}(M_1, M_1(x), M_2(x))\\
&\mathit{semicolon}(M, C_\alpha, C'_{must}) = \mathit{update}(M, C')_{must}\\
&\mathit{semicolon}(M, C_\alpha, C'_{sur}) = (\mathit{update}(M, C') \cup C)_\alpha
\end{aligned}$$

The function $\mathit{update}(M, C)$ checks the constant map M for each variable found in the elements of the set C, and if there exists a mapping in M for that variable, uses it to update C. For example, if $M(y) = \{w, z\}$ and $C = \{y + 1\}$, $\mathit{update}(M, C)$ returns $\{w + 1, z + 1\}$.

The **abs** function, where $i \in \mathbb{Z}$, is

$$\begin{aligned}
\mathbf{abs}(M) = \lambda S.\lambda x. &\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(x)_{must}, M(x))\\
&\text{in if } C = \{i\} \text{ then } i \text{ else } \bot
\end{aligned}$$

Theorem 4.2.5. *R for* CP *is an exact representation.*

Proof. Given in the appendix. □

[1] Precise definitions of *isConstant* and *consVal* depend on the kind of constant propagation chosen (e.g. literal, copy, linear, or non-linear constant propagation [SRH96]).

4.2.5 Loop Invariants (LI)

We take the definition of a loop invariant as given in [ASU86]:

> A statement inside a loop L is invariant if all the operands of the statement either are constant, have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is an invariant statement in L.

As a simplification we will compute the invariance information of a statement only with respect to the innermost enclosing loop that surrounds the statement. We assume that there exists a function $loop(P)$ to obtain that innermost loop. We also assume that the reaching definitions have been computed and are available for use in LI: $\mathsf{RD}(n, y)$ gives the definitions of y that reach the node n; $\mathsf{RD}_R(n)$ gives the RD representation for the node n. (Alternatively, RD can be computed on-the-fly.)

$Data$ is defined as a map containing the invariance information:

$$I \in Data = (Node \rightharpoonup Bool) \cup \{\top\}$$

$Data$ is ordered as follows:

$$\begin{aligned}&I \sqsubseteq I' \text{ if } \forall n.I'(n) \text{ is undefined or}\\ &\qquad\qquad I'(n) \sqsubseteq I(n) \text{ in the boolean lattice, where false } \sqsubset \text{ true}\end{aligned}$$

The definitions of *exp* and *asgn* are

$$\begin{aligned}&exp(e) = id\\ &asgn(n, x, e) = \lambda I.I[n \mapsto \forall y \in vars(e).\mathsf{isInv}(n, y, I)]\end{aligned}$$

where isInv is defined as

$$\begin{aligned}\mathsf{isInv}(n, y, I) = {} & loop(n) \text{ is defined and}\\ & ((\forall d \in \mathsf{RD}(n, y)\,.\, d \text{ is not contained in } loop(n))\\ & \vee(\exists d\,.\,\mathsf{RD}(n, y) = \{d\} \text{ and } I(d)))\end{aligned}$$

isInv directly follows from the definition of loop invariants: Invariance of a node is dependent on (1) the reaching definitions of a variable, or (2) a single node if there is only one reaching definition. This also hints at the definition of a representation:

$$\begin{aligned}R &= (Node \rightharpoonup IV) \cup \{\top_R\}\\ IV &= \mathcal{P}(Var \cup Node \cup \{\text{true}, \text{false}\})\end{aligned}$$

The invariance information we keep per node, called IV, is a set that contains variables, nodes, true, or false, where a variable stands for the dependence (1), and a node stands for the dependence (2). The intuition is that if the IV set of a statement contains a node, that node must become invariant for the statement to be invariant; if the set contains a variable, the reaching definitions of that variable must eventually satisfy the conditions for making the statement an invariant. If the available information is enough to conclude that the statement is not invariant, the set contains false; if the only item in the set is true, the statement is invariant. In other words, the IV is used to delay the computation of invariance of a statement. As more information becomes available, IV is updated. Below are the necessary definitions.

$$
\begin{aligned}
&\mathit{exp}_R(e) = \mathit{id}_R \\
&\mathit{asgn}_R(x, e, n) = \{n \mapsto vars(e) \cup \{\mathsf{true}\}\} \\
&M_1 \wedge_R M_2 = \lambda n.\, M_1(n) \cup M_2(n) \\
&M_1 ;_R M_2 = M_1 \uplus fix\big(\lambda M_2'.\lambda n\,.\,\{\mathsf{update}_n(s, M_1 \uplus M_2') | s \in M_2(n)\}\big)
\end{aligned}
$$

where $\uplus$ is domain disjoint union of functions, and update is defined as

$\mathsf{update}_n(\text{true}, M) = \text{true}$
$\mathsf{update}_n(\text{false}, M) = \text{false}$
$\mathsf{update}_n(n', M) =$ if $M(n') = \{\text{true}\}$ // *n' is invariant*
  then true
  else if false $\in M(n')$ // *n' is not invariant*
    then false
    else n' // *cannot update yet, so keep n'*
$\mathsf{update}_n(x, M) =$ let $(K, G) \leftarrow \mathsf{RD}_R(n)$ in
  if there are no definitions of x in G
  then x // *cannot update yet, so keep x*
  else if there are multiple definitions of x in G
    then if all the definitions are outside $\mathit{loop}(n)$
      then true // *any definition that may become available later on*
        // *is guaranteed to be outside the loop as well*
      else false
  else if there is a single definition d of x in G
    then if d is outside $\mathit{loop}(n)$ then true
      else if $x \in K$
        then if $M(d) = \{\text{true}\}$ then true else d
        else false

Finally, we give the definition of the **abs** function

$$
\mathbf{abs}(M) = \lambda I.\, I \wedge I' \text{ where } I' \text{ is}
$$
$$
fix\left(\begin{array}{l} \lambda I_2.\lambda n.\ \text{let } B \leftarrow \{\mathsf{isInv}_R(n, s, I \wedge I_2) | s \in M(n)\} \\ \qquad\qquad \text{in } B = \{\text{true}\} \end{array} \right)
$$

and isInv_R is a function that returns a boolean value:

$$\begin{aligned}
&\mathsf{isInv}_R(n, \text{true}, I) = \mathit{loop}(n) \text{ is defined}\\
&\mathsf{isInv}_R(n, \text{false}, I) = \text{false}\\
&\mathsf{isInv}_R(n, n', I) = \mathit{loop}(n) \text{ is defined and } I(n')\\
&\mathsf{isInv}_R(n, x, I) = \mathsf{isInv}(n, x, I)
\end{aligned}$$

Note that there is recursion in the definitions of $;_R$ and **abs**. The recursion terminates because it is not possible to have in a valid program two nodes which are solely dependent on each other, or a node whose invariance is only dependent on itself.

4.2.6 Type Checking (TC)

Type-checking can also be staged by defining a representation. It also extends the language with declarations and scope to be useful. Michael Katelman gives a detailed analysis of this in his MS thesis [Kat06].

4.3 Framework for Backward Analysis

We can define a similar framework for backwards analysis. We directly start with the intermediate framework that contains break statements. To illustrate how the framework would be affected by the change in the direction of the dataflow, let us first look at the diagrams in Figure 4.12 where the dataflow for a labelled statement containing break statements is depicted for both forward and backward direction. Recall that in the forward framework, we used the environment to carry the data that flow on the break edges. In the backward framework, we use the environment for the same purpose. However, this time, as the diagram in Figure 4.12 also indicates, the datum on a break edge is an *input* into the node, rather than an *output*. For a labelled statement $\ell : P$, recall that the forward framework took the meet of data that comes along the break edges (i.e. abnormal exits from P to label ℓ) and the normal exit from P at the exit of the labelled statement.

In backward dataflow, note that the incoming data of P —the body of the labelled statement— come not only from the normal execution path, but also along the break edges. Thus, we need to put the input data of a labelled statement directly into the environment before analyzing the body P; there is no meet operation. For a break statement, there cannot be any data coming from the normal execution path — no such path exists. The incoming data should be taken directly from the environment, and forwarded as the output data. The dataflow for other syntactic constructs is straightforward. The formal definition of the backward analysis framework is given in Figure 4.13.

Computation of representations for the backward analysis functions is presented in Figure 4.14. The definition for the break statement and labelled statement deserves an

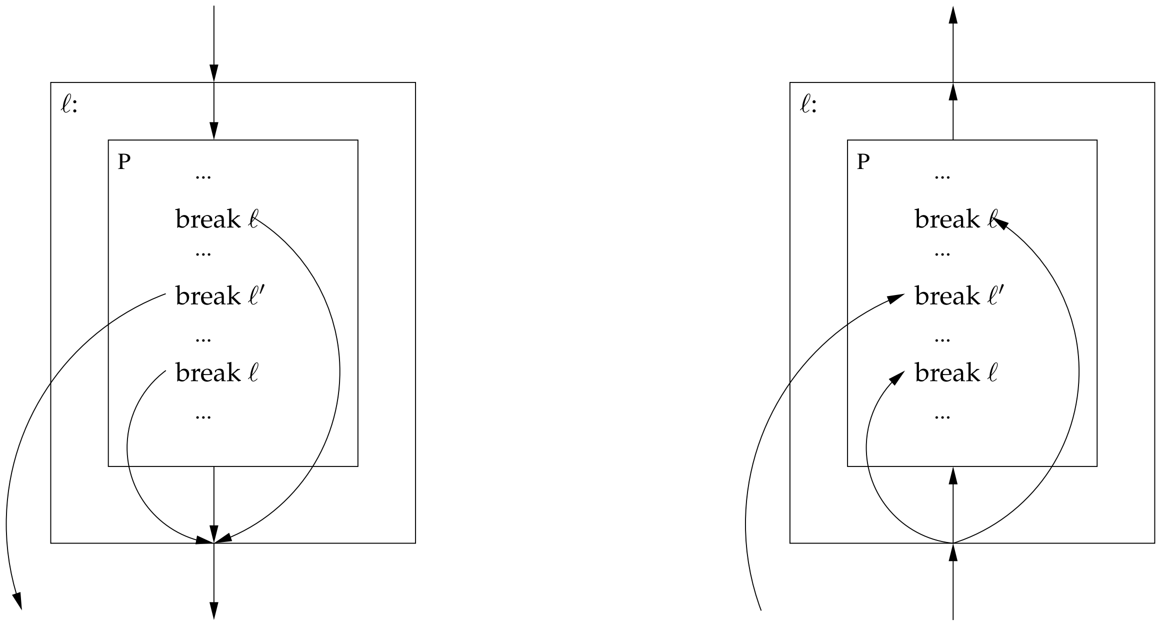

Figure 4.12: The flow of data for the labelled statement $\ell : P$ in the forward and backward direction. Note that in backward analysis, there may be multiple entrance points into a node as opposed to single entrance point in the forward analysis.

explanation; the definition of other syntactic constructs is straightforward. The function $\mathcal{R}[\![P]\!]$ returns a pair of an environment and a representation similar to the one for forward analysis. The second item of this pair (i.e. the representation) stands for the dataflow function along the normal execution path. The first item (i.e. the environment) contains for each label a representation which stands for the meet of dataflow functions along the break edges for the corresponding label. The function $\mathcal{R}[\![P]\!]$ returns $(\top_{Env_R}[\ell \mapsto id_R],\ \top_R)$ for a break statement, because a break statement does not have a normal execution path (hence $\top_R$ is returned as the representation) and it forwards any incoming data along its break edge without any modification (hence $\top_{Env_R}[\ell \mapsto id_R]$ is returned as the environment). For a labelled statement $\ell : P$, the environment that comes from the analysis of the body P is updated to map the label ℓ to $\top_R$ because all the break edges for the label ℓ are enclosed within the labelled statement; there are no dangling break edges for label ℓ. Now that we know the scope of the label, we take the meet of the representation of the body and the representation for the label, and return it as the second item of the pair. In other words, the data that arrives at the exit of the labelled statement are the data that flow along the normal execution path plus the data that flow along the break edges.

The abstraction function for the intermediate backward framework is

$$\mathbf{abs}_E(\eta_R, r) = \lambda(\eta, d).(\eta, \mathbf{abs}(r)(d) \wedge \bigwedge_{\ell \in \mathrm{Label}} \mathbf{abs}(\eta_R(\ell))(\eta(\ell)))$$

$$
\begin{aligned}
\mathcal{B}[\![\mathsf{skip}]\!] &= id \\
\mathcal{B}[\![x = e]\!] &= \lambda(\eta, d).(\eta, asgn(x, e)(d)) \\
\mathcal{B}[\![\mathsf{break}\ \ell]\!] &= \lambda(\eta, d).(\eta, \eta(\ell)) \\
\mathcal{B}[\![\ell : P]\!] &= \lambda(\eta, d).\ \text{let } (\eta', d') \leftarrow \mathcal{B}[\![P]\!](\eta[\ell \mapsto d], d) \\
&\qquad\qquad\quad \text{in } (\eta'[\ell \mapsto \top_{Data}], d') \\
\mathcal{B}[\![P_1; P_2]\!] &= \mathcal{B}[\![P_2]\!]; \mathcal{B}[\![P_1]\!] \\
\mathcal{B}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] &= \lambda(\eta, d).\ \text{let } (\eta_1, d_1) \leftarrow \mathcal{B}[\![P_1]\!](\eta, d) \\
&\qquad\qquad\qquad\quad (\eta_2, d_2) \leftarrow \mathcal{B}[\![P_2]\!](\eta, d) \\
&\qquad\qquad\quad \text{in } (\eta, exp(e)(d_1 \wedge d_2))
\end{aligned}
$$

Figure 4.13: Intermediate framework for backward analysis.

$$
\begin{aligned}
\mathcal{R}[\![\mathsf{skip}]\!] &= (\top_{Env_R},\ id_R) \\
\mathcal{R}[\![x = e]\!] &= (\top_{Env_R},\ asgn_R(x, e)) \\
\mathcal{R}[\![\mathsf{break}\ \ell]\!] &= (\top_{Env_R}[\ell \mapsto id_R],\ \top_R) \\
\mathcal{R}[\![\ell : P]\!] &= \text{let } (\eta, r) \leftarrow \mathcal{R}[\![P]\!] \\
&\quad\ \text{in } (\eta[\ell \mapsto \top_R],\ r \wedge_R \eta(\ell)) \\
\mathcal{R}[\![P_1; P_2]\!] &= \text{let } (\eta_1, r_1) \leftarrow \mathcal{R}[\![P_1]\!], (\eta_2, r_2) \leftarrow \mathcal{R}[\![P_2]\!] \\
&\quad\ \text{in } (\eta_1 \wedge_R (\eta_2 ;_R r_1),\ r_2 ;_R r_1) \\
\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] &= (\mathcal{R}[\![P_1]\!] \wedge_R \mathcal{R}[\![P_2]\!]) ;_R exp_R(e)
\end{aligned}
$$

Figure 4.14: Representation for framework of Figure 4.13.

Theorem 4.3.1. *For a legal program P, if the DFFun functions are distributive (i.e. $f(d \wedge d') = f(d) \wedge f(d')$), then $\mathbf{abs}_E(\mathcal{R}[\![P]\!]) = \mathcal{B}[\![P]\!]$.*

Proof. By induction on the structure of P. □

For the full framework which builds a node map at the top node, the intermediate framework can again be extended naturally as in forward analysis (Figure 4.9). However, defining $\mathcal{R}$ is not that straightforward. We need to keep an environment for every node in the node-map. This is because in the backward analysis, there may be multiple entrance points into a node as opposed to a single entry in the forward analysis; see Figure 4.12 again for an example. The representation we keep associated to a node stands for the dataflow function when the data enter from the normal execution path; the environment keeps the representations that stand for the abnormal execution paths (i.e. along break edges). So the type of the function $\mathcal{R}$ is

$$\mathcal{R} : \text{Pgm} \rightarrow (Node \rightarrow (Env_R \times R)) \times Env_R \times R$$

$$
\begin{aligned}
&\mathcal{B}[\![n : \mathsf{skip}]\!] = \mathit{id}\\
&\mathcal{B}[\![n : x = e]\!] = \lambda(\varphi,\eta,d).(\varphi[n \mapsto \mathit{asgn}(x,e)(d)],\eta,\mathit{asgn}(x,e)(d))\\
&\mathcal{B}[\![n : \mathsf{break}\ \ell]\!] = \lambda(\varphi,\eta,d).(\varphi[n \mapsto \eta(\ell)],\eta,\eta(\ell))\\
&\mathcal{B}[\![n : (\ell : n_1 : P)]\!] = \lambda(\varphi,\eta,d).\ \text{let } (\varphi',\eta',d') \leftarrow \mathcal{B}[\![n_1 : P]\!](\varphi,\eta[\ell \mapsto d],d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } (\varphi'[n \mapsto d'],\eta'[\ell \mapsto \top_{Data}],d')\\
&\mathcal{B}[\![n : (n_1 : P_1; n_2 : P_2)]\!] = \lambda(\varphi,\eta,d).\ \text{let } (\varphi',\eta',d') \leftarrow (\mathcal{B}[\![n_2 : P_2]\!];\mathcal{B}[\![n_1 : P_1]\!])(\varphi,\eta,d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } (\varphi'[n \mapsto d'],\eta',d')\\
&\mathcal{B}[\![n : \mathsf{if}(e)\ n_1 : P_1\ \mathsf{else}\ n_2 : P_2]\!] = \lambda(\varphi,\eta,d).\ \text{let } (\varphi_1,\eta_1,d_1) \leftarrow \mathcal{B}[\![n_1 : P_1]\!](\varphi,\eta,d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\varphi_2,\eta_2,d_2) \leftarrow \mathcal{B}[\![n_2 : P_2]\!](\varphi,\eta,d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in } ((\varphi_1 \cup \varphi_2)[n \mapsto \mathit{exp}(e)(d_1 \wedge d_2)],\eta,\mathit{exp}(e)(d_1 \wedge d_2))
\end{aligned}
$$

Figure 4.15: Full framework for backward analysis.

$$
\begin{aligned}
&\mathcal{R}[\![n : \mathsf{skip}]\!] = (\{n \mapsto (\top_{Env_R},\mathit{id}_R)\},\ \top_{Env_R},\ \mathit{id}_R)\\
&\mathcal{R}[\![n : x = e]\!] = (\{n \mapsto (\top_{Env_R},\mathit{asgn}_R(x,e))\},\ \top_{Env_R},\ \mathit{asgn}_R(x,e))\\
&\mathcal{R}[\![n : \mathsf{break}\ \ell]\!] = (\{n \mapsto (\top_{Env_R}[\ell \mapsto \mathit{id}_R],\top_R)\},\ \top_{Env_R}[\ell \mapsto \mathit{id}_R],\ \top_R)\\
&\mathcal{R}[\![n : (\ell : n_1 : P)]\!] = \text{let } (\varphi,\eta,r) \leftarrow \mathcal{R}[\![n_1 : P]\!]\\
&\qquad\qquad\qquad\qquad \text{in } (\mathit{closeLabel}(\ell,\ \varphi[n \mapsto (\eta,r)]),\ \eta[\ell \mapsto \top_R],\ r \wedge_R \eta(\ell))\\
&\mathcal{R}[\![n : (n_1 : P_1; n_2 : P_2)]\!] = \text{let } (\varphi_1,\eta_1,r_1) \leftarrow \mathcal{R}[\![n_1 : P_1]\!], (\varphi_2,\eta_2,r_2) \leftarrow \mathcal{R}[\![n_2 : P_2]\!]\\
&\qquad\qquad\qquad\qquad\qquad \text{in } (\lambda n'.\ \text{if } \varphi_2(n') \text{ defined then } \varphi_2(n')\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{if } \varphi_1(n') \text{ defined then let } (\eta',r') \leftarrow \varphi_1(n')\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in } (\eta' \wedge_R (\eta_2;_R r'),\ r_2;_R r')\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{if } n' = n \text{ then } (\eta_1 \wedge_R (\eta_2 ;_R\ r_1),\ r_2 ;_R\ r_1),\\
&\qquad\qquad\qquad\qquad\qquad \eta_1 \wedge_R (\eta_2 ;_R\ r_1),\ r_2 ;_R\ r_1)\\
&\mathcal{R}[\![n : \mathsf{if}(e)\ n_1 : P_1\ \mathsf{else}\ n_2 : P_2]\!] = \text{let } (\varphi_1,\eta_1,r_1) \leftarrow \mathcal{R}[\![n_1 : P_1]\!], (\varphi_2,\eta_2,r_2) \leftarrow \mathcal{R}[\![n_2 : P_2]\!]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{in } ((\varphi_1 \cup \varphi_2)[n \mapsto (r_1 \wedge_R r_2);_R \mathit{exp}_R(e)],\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (\eta_1 \wedge_R \eta_2);_R \mathit{exp}_R(e),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (r_1 \wedge_R r_2);_R \mathit{exp}_R(e))
\end{aligned}
$$

Figure 4.16: Representation for framework of Figure 4.15.

Analogous to how $\mathcal{R}[\![P_1; P_2]\!]$ in the forward representation function of Figure 4.10 updates the node-map for each node in P_1 and P_2, $\mathcal{R}[\![\ell : P]\!]$ and $\mathcal{R}[\![P_1; P_2]\!]$ in the full backward representation function update each mapping in their node-maps as well. Full versions of $\mathcal{B}$ and $\mathcal{R}$ are given in Figures 4.15 and 4.16, respectively.

In Figure 4.16, *closeLabel* is defined as

$$\mathit{closeLabel}(\ell, \varphi) = \lambda n.\, \text{let } (\eta, r) \leftarrow \varphi(n) \text{ in } (\eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))$$

The **abs** function for the full backward framework is defined as

$$\begin{aligned}\mathbf{abs}_F(\varphi, \eta, r) = \lambda(\varphi', \eta', d').\, \text{let } \varphi'' \leftarrow \lambda n.\, &\text{let } (\overline{\eta}, \overline{r}) \leftarrow \varphi(n) \\ &\quad \text{in } \mathbf{abs}(\overline{r})(d') \wedge \textstyle\bigwedge_{\ell \in \text{Label}} \mathbf{abs}(\overline{\eta}(\ell))(\eta'(\ell)) \\ \text{in } (\varphi' \cup \varphi'',\ \eta',\ &\mathbf{abs}(r)(d') \wedge \textstyle\bigwedge_{\ell \in \text{Label}} \mathbf{abs}(\eta(\ell))(\eta'(\ell)))\end{aligned}$$

Theorem 4.3.2. *For a legal program P, if the DFFun functions are distributive (i.e. $f(d \wedge d') = f(d) \wedge f(d')$), then $\mathbf{abs}_F(\mathcal{R}[\![P]\!]) = \mathcal{B}[\![P]\!]$.*

Proof. Similar to the proof for the intermediate framework (Theorem 4.3.1). $\square$

4.3.1 Live Variables (LV)

Data is defined as

$$L \in \mathit{Data} = (\mathcal{P}(\text{Var})) \cup \{\top\}$$

and is ordered by reverse set inclusion.

$$\begin{aligned}&\mathit{asgn}(n, x, e) = \lambda L.(L \setminus \{x\}) \cup \mathit{vars}(e) \\ &\mathit{exp}(e) = \lambda L.L \cup \mathit{vars}(e)\end{aligned}$$

$$\begin{aligned}&R = \mathcal{P}(\text{Var})^2 \\ &\mathit{asgn}_R(n, x, e) = (\{x\}, \mathit{vars}(e)) \\ &\mathit{exp}_R(e) = (\varnothing, \mathit{vars}(e))\end{aligned}$$

Definitions of id_R, $;_R$, $\wedge_R$ and **abs** are the same as in RD (Section 4.2.1).

Note that LV is a distributive analysis.

4.3.2 Very Busy Expressions (VBE)

The definitions, except the following, are the same as in AE.

$$\begin{aligned}&\mathit{asgn}(n, x, e) = \lambda E.(E \setminus E_x) \cup \mathit{sub}(e) \\ &\mathit{asgn}_R(n, x, e) = (\{x\}, \{e'_{\mathit{must}} \mid e' \in \mathit{sub}(e)\})\end{aligned}$$

Note that VBE is a distributive analysis.

	HotSpot			libgcj			Kaffe		
Sample Program	RD	CP	TC	RD	CP	TC	RD	CP	TC
Big-plug	2.10	1.19	3.65	7.43	3.78	5.15	9.73	5.23	5.63
Small-plug-A	2.17	1.12	3.50	6.96	3.91	4.28	10.7	4.62	5.55
Small-plug-B	2.40	1.14	2.97	4.78	3.41	4.39	7.03	4.65	5.40
Two-plug	1.67	1.17	1.66	2.59	2.19	2.90	3.83	2.83	3.18
Fib1 ([Kam04])	1.10	1.07	1.31	1.24	0.93	1.17	1.64	1.26	1.05
Fib2 ([Kam04])	1.23	1.16	0.67	1.48	0.99	1.18	2.02	1.47	1.05
Sort ([KCC00a])	1.48	1.21	1.92	1.64	1.08	1.59	1.86	1.29	1.66
Huffman ([Kam04])	1.11	1.29	0.30	1.04	0.93	1.02	1.31	1.30	0.95
Marshalling 1 ([AJKC05])	12.37	3.93	28.27	34.83	15.42	9.34	49.64	18.92	12.04
Marshalling 2 ([AJKC05])	2.01	1.75	16.01	1.83	1.33	1.86	2.59	2.27	1.47

Table 4.1: Benchmarking results. The numbers show the speedup factor: ratio of the *base case* to the *staged* case.

4.4 Performance

We are interested in the *run-time* costs of two methods of doing static analysis. One method is to fill in the holes and analyze the complete program at run time (the *base analysis*); the other is to use our *staged analysis*.

The benchmarks we present are of two kinds: *artificial* benchmarks illustrate how performance is affected by specific features in a program; *realistic* benchmarks are program generators drawn from previous publications.

For some analyses, one needs only the dataflow information for the root node; examples are uninitialized variables and type-checking. For most, we need the information at many, though not necessarily all, nodes. (Note that the base case must visit every node at run-time, even if it is only interested in a subset.)

We implemented the framework in Java. In Table 4.1, we present the performance of three analyses, on a variety of benchmark programs, as ratios between the base and the staged analyses; higher numbers represent greater speed-up. We run the experiments in three different Java runtime environments: Sun's HotSpot (with the client setting), GNU's libgcj, and Kaffe. For reaching definitions (RD) and constant propagation (CP), we perform the analysis at every assignment statement (roughly half the nodes in the programs). For type checking (TC), we analyze only the top node. Benchmarking was done on a Linux machine with 1.5 GHz CPU and 1GB memory.

We briefly describe the benchmarks used in Table 4.1.

- **Big-plug** is a small program with one hole, filled in by a large plug.
- **Small-plug-A** is a large program with a hole near the beginning, filled in by a small plug.

- **Small-plug-B** is a large program with a hole near the end, filled in by a small plug.
- **Two-plug** is a medium-sized program with two holes, filled in by medium-sized plugs.
- **Fib1** and **Fib2** are two versions of a Fibonacci function divided into small pieces for exposition [Kam04].
- **Sort** is a generator that produces a sort function by inlining the comparison operation [KCC00a].
- **Huffman** is a generator that turns a Huffman tree into a sequence of conditional statements [Kam04].
- **Marshalling 1** is part of a program that produces customized serializers in Java [AJKC05]; characteristics much like Big-plug.
- **Marshalling 2** is a different part of the same program; it has many holes and many small plugs.

As often happens, the invented benchmark examples show the best performance improvements. Our approach does result in slow-downs in some cases; the worst cases are Fib2 and Huffman, both of which consist of many holes and small plugs. Overall, the results are quite promising.

4.5 Related Work

Our work shares with several others a concern with *representation* of dataflow functions, and some of our representations have appeared previously. In the area of *interprocedural dataflow analysis*, Sharir and Pnueli [SP81] introduced the idea of summarizing the analysis of an entire procedure. Rountev, Kagan and Marlowe [RKM06] discuss concrete representations for these summary functions, to allow for "whole program" analysis of programs that use libraries; our representation for reaching definitions (RD) appears there. Reps, Horwitz, and Sagiv [RHS95] give representations for a class of dataflow problems, including reaching definitions and linear constant propagation. (Interprocedural analysis is similar to staged analysis in that one can think of the procedure call as a "hole," and the procedure as a "plug." However, the control flow issues are very different; that work must deal with the notion of "valid" paths — where calls match returns — while we must deal with multiple-exit control structures.) To parallelize static analyses, Kramer, Gupta and Soffa [KGS94] partition programs and analyze each partition to produce a summary of its effect on the program as a whole.

In *hybrid* analysis [MR90], Marlowe and Ryder partition a program based on strong components, representing dataflow functions for each component. A representation for reaching definitions that is "adequate" in our sense is given there. Marlowe and Ryder also talk about *incremental analysis* where the problem is to maintain the validity of an analysis during source program editing. But note the subtle but important distinction between *incremental* analysis and *staged* analysis: there, *any* node can change at any time; here, some parts of the program are fixed and some unknown, and the goal is to fully exploit the fixed parts.

In *approximate analysis* [SGM+03], the meta-program is analyzed to determine as much as possible about what the generated program will look like. This approach has the advantage of avoiding run-time analysis entirely, but the disadvantage that the analysis results are very approximate.

Lastly, we mention the work of Chambers et al. [Cha02]. That work has the ambitious goal of *automatically* staging compilers: a user can indicate when some information will first become available, and the system will produce an optimizer to *efficiently* perform the optimization at that time. The broad goals of that work — optimizing run-time compilation — are the same as ours. However, we are much less ambitious about the use of automation (and, indeed, that work accommodates a limited number of optimizations); we are, instead, providing a mathematical framework that can facilitate the manual construction of staged analyses.

4.6 Conclusions

We have presented a framework for static analysis of ASTs, including break statements, that allows the analysis to be staged, when the representations are adequate. The method has application to run-time program generation: by optimizing the static analysis of programs, it can speed up overall run-time code generation time. We presented representations for several data-flow analyses, namely reaching definitions, available expressions, constant propagation, loop invariance, live variables, and very busy expressions. We provided experimental results to demonstrate that staging can achieve significant runtime performance improvement. The technique has not been integrated into a program generation system; we leave this as a future work.

Chapter 5

Record Calculus as a Staged Type System

In this chapter we focus on the second challenge of program generation: How can we guarantee that a generator will produce type-safe code? Several program generation type systems investigate the same question [Dav96, DP96, KKcS08, KYC06, MTBS99, OMY01, Rhi05, SGM$^+$03, TN03, YI06]. We show that this problem reduces to the problem of type checking in record calculus, which is well-studied and mature. This allows us to reuse several properties already proved in the record calculus domain, giving us a powerful and sound type system that guarantees type-safety of generated programs.

Major results in this chapter include:

- Definition of a translation from a program generation language to the record calculus.
- Showing that evaluating a program generator in the staged operational semantics is equivalent to evaluating its translation in the record operational semantics. This result brings the preservation property to the record type system with respect to the staged semantics *for free*.
- Proving that the record calculus provides a sound type system with respect to the staged operational semantics.
- Showing that the record calculus type system is equal to the λ^{open}_{poly} [KYC06] type system.

We then show that

- the type system can gracefully be extended with subtyping constraints by using already-existing record subtyping definitions from the literature. A staged type system with subtyping constraints, to our knowledge, is new.
- pluggable declarations can be added to the language. Pluggable declarations and subtyping provide a solution to type-checking the "library specialization problem" [AK09].
- side-effecting expressions such as references can also be handled by an improved version of the translation.

The results elaborated in this chapter show that a very powerful staged type system can be obtained by using record calculus properties.

This chapter is organized as follows: In Section 5.1 we give intuition of why record calculus and program generation are closely related. Section 5.2 informally discusses how a staged type system works and what we expect from it, and motivate the need for subtyping. We formally introduce the program generation language and the record calculus in Sections 5.3 and 5.4, respectively. Section 5.5 gives the definition of the translation from the staged language to the record calculus. Section 5.6 states the formal relationship between the two calculi. In Sections 5.7 through 5.9 we discuss how to extend the languages, translation, and the type system with subtyping, pluggable declarations, and references. We provide a comparison of our contributions to the existing work in Section 5.10. We conclude the chapter in Section 5.11. Proofs of major lemmas and theorems of this chapter are given in the appendix.

5.1 Using Records for Staged Computing

A quotation defines a program piece that is not executed until it is "run". Consider $\langle 2+3\rangle$. This expression directly evaluates to the value $\langle 2{+}3\rangle$, not $\langle 5\rangle$. "2+3" is executed only when the quoted expression is "run" as in $\mathsf{let}\, x = \langle 2+3\rangle \,\mathsf{in}\, \mathsf{run}(x)$, which evaluates to 5. This fact brings a question about the relation between quoted expressions and closures. Recall that in almost any programming language, expressions guarded by lambda abstractions are not executed until the function is applied. We can represent a quoted expression as a lambda abstraction, and "run" as function application. $\langle 2+3\rangle$ can be represented as $\lambda z.2+3$, which directly evaluates to the closure $\lambda z.2+3$ without executing $2+3$. So, we can rewrite $\mathsf{let}\, x = \langle 2+3\rangle \,\mathsf{in}\, \mathsf{run}(x)$ as $\mathsf{let}\, x = \lambda z.2+3 \,\mathsf{in}\, x(0)$, where the application evaluates $2+3$ and results in 5. The name of the function parameter and the function argument are not important in this example. So, we have an indication that there is a close relation between lambda abstractions and quoted expressions, as well as "run" and function application.

Let us now consider a more complicated example which splices a fragment into another one using antiquotation: $\mathsf{let}\, x = \langle 2+3\rangle \,\mathsf{in}\, \langle 4+ \text{`}(x)\rangle$. This piece of program evaluates to $\langle 4+(2+3)\rangle$. The body of the quoted expression $\langle 2+3\rangle$ is still not executed, but "extracted out" and spliced into the hole as denoted by the antiquotation. To give a similar effect, we can consider converting the antiquotation to function application: $\mathsf{let}\, x = \lambda z.2+3 \,\mathsf{in}\, \lambda w.4+x(0)$. Because the function application takes place under a lambda abstraction, the expression "$2+3$" is still not executed, until the context is.

The two examples above were simple in the sense that the quoted expressions were closed; they did not contain any free variables. Consider $\langle y\rangle$. The variable y will obtain a meaning when the expression is spliced into a context that provides a binding for y. For instance, in $\mathsf{let}\, c = \langle y\rangle \,\mathsf{in}\, \langle \mathsf{let}\, y = 2 \,\mathsf{in}\, \text{`}(c)+3\rangle$, the variable y is an integer. This bind-

$$\llbracket c \rrbracket^n = c$$
$$\llbracket x \rrbracket^n = r_n \cdot x$$
$$\llbracket \lambda x.e \rrbracket^n = \lambda x.\, \mathsf{let}\, r_n = r_n\, \mathsf{with}\, \{x = x\}\, \mathsf{in}\, \llbracket e \rrbracket^n$$
$$\llbracket \mathsf{fix}\, f(x).\, e \rrbracket^n = \mathsf{fix}\, f(x).\, \mathsf{let}\, r_n = r_n\, \mathsf{with}\, \{f = f, x = x\}\, \mathsf{in}\, \llbracket e \rrbracket^n$$
$$\llbracket e_1 e_2 \rrbracket^n = \llbracket e_1 \rrbracket^n \llbracket e_2 \rrbracket^n$$
$$\llbracket \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \rrbracket^n = \mathsf{let}\, r_n = r_n\, \mathsf{with}\, \{x = \llbracket e_1 \rrbracket^n\}\, \mathsf{in}\, \llbracket e_2 \rrbracket^n$$
$$\llbracket \langle e \rangle \rrbracket^n = \lambda r_{n+1}.\, \llbracket e \rrbracket^{n+1}$$
$$\llbracket \grave{}(e) \rrbracket^{n+1} = \llbracket e \rrbracket^n\, r_{n+1}$$
$$\llbracket \mathsf{run}(e) \rrbracket^n = \llbracket e \rrbracket^n \{\}$$
$$\llbracket \mathsf{lift}(e) \rrbracket^n = \lambda r_{n+1}. \llbracket e \rrbracket^n$$

Figure 5.1: A first attempt on a transformation from staged expressions to record calculus expressions. Variable names are also used as record field labels in the target language. The superscript n in the translation denotes the stage. The environment at stage n is represented with the record variable r_n.

ing is provided by the code fragment surrounding the antiquotation. Hence, it makes sense to consider a quoted expression as a lambda abstraction that takes in the bindings of its free variables rather than ignoring the parameter. The "bindings" are nothing but an environment. An occurrence of a variable is then a lookup in the environment. So, using the dot notation $e \cdot \ell$ to access the field ℓ of the record e, we can rewrite $\langle y \rangle$ as $\lambda r.r \cdot y$. Note that quoted expressions also can define and use variables within themselves. These bindings can be considered as updates to the environment. So, an antiquotation becomes a function application that passes the up-to-date environment to the antiquoted fragment. The program above, $\mathsf{let}\, c = \langle y \rangle\, \mathsf{in}\, \langle \mathsf{let}\, y = 2\, \mathsf{in}\, \grave{}(c) + 3 \rangle$, can be rewritten as $\mathsf{let}\, c = (\lambda r.r \cdot y)\, \mathsf{in}\, \lambda r.\mathsf{let}\, r = r\, \mathsf{with}\, \{y = 2\}\, \mathsf{in}\, c(r) + 3$, where $e\, \mathsf{with}\, \{\ell = e'\}$ is the operation that returns a record exactly the same as e, with the exception that field ℓ maps to e'. The "run" operation is also a function application similar to an antiquotation, but it should pass the empty environment to the fragment, because only complete (i.e. closed) fragments are runnable.

The intuitive closeness between the staged language and the record calculus immediately suggests a systematic translation. In Figure 5.1, we give a transformation to convert staged expressions to record calculus expressions. The superscript n in the translation denotes the stage. The environment at stage n is represented with the record variable r_n. Note that the environment r_n is simply a variable; e.g. r_0 is the environment of the meta level, r_1 is the environment of the first stage. Assuming that record variables do not exist in the staged calculus, we do not get any name collision. The translation converts a variable x to a look-up in the environment of the current stage, denoted $r_n \cdot x$. Abstractions and let-bindings update the current environment with a new binding. In this update, the bound variable name is used as the field name in the environment. Quoted expressions are converted to functions that take as input a record representing the environment in the next stage. Antiquotations are translated to function applications where the current environment becomes the operand. $\mathtt{run}(\cdot)$ is also converted to a function application, but this time the operand is the empty record. $\mathsf{fix}\, f(x)$ is the fix-point operator for the function f with argument x, and is used for recursion. lift raises a value to the next stage; hence the translation introduces an abstraction.

We give this first attempt of a translation; however, we will not use it in the upcoming sections. We will define better versions; the first improvement will provide more useful results in proving formal properties, the second improvement will handle updatable references as well. We initially give this version because it is more intuitive and easier to understand than the other improved versions. Below we give some examples to illustrate the translation process. The reader is encouraged to check that both sides would reduce to equivalent terms when simplified (after substituting r_0 with the empty record in translations). To improve readability of the examples, we assume existence of constructs such as the if-expression, lists, head (hd) and tail (tl) operators, addition, subtraction, etc. It would be straightforward to add these into the language. All the translations take place starting from stage 0 as denoted by the superscript.

$$
\begin{array}{rl}
[\![\lambda c.\langle \mathsf{let}\, x = 5\, \mathsf{in}\, \grave{}(c)\rangle]\!]^0 = & \lambda c.\mathsf{let}\, r_0 = r_0\, \mathsf{with}\, \{c = c\}\, \mathsf{in} \\
& \quad (\lambda r_1.\mathsf{let}\, r_1 = r_1\, \mathsf{with}\, \{x = 5\}\, \mathsf{in}\, r_0{\cdot}c(r_1))
\end{array}
$$

$$
\begin{array}{lcl}
[\![\mathsf{let}\, c = \langle x + 8\rangle\, \mathsf{in} & & \mathsf{let}\, r_0 = r_0\, \mathsf{with}\, \{c = \lambda r_1.r_1{\cdot}x + 3\}\, \mathsf{in} \\
\;\mathsf{let}\, g = \langle \lambda x.\grave{}(c)\rangle\, \mathsf{in} & = & \mathsf{let}\, r_0 = r_0\, \mathsf{with}\, \{g = \lambda r_1.\lambda x.\mathsf{let}\, r_1 = r_1\, \mathsf{with}\, \{x = x\}\, \mathsf{in}\, r_0{\cdot}c(r_1)\}\, \mathsf{in} \\
\;(\mathsf{run}(g))(10)]\!]^0 & & (r_0{\cdot}g(\{\}))(10)
\end{array}
$$

The function below is the factorial function and is written completely in stage 0.

$$
\begin{array}{ll}
[\![\mathsf{fix}\, fact(n).\, \mathsf{if}\, n = 0\, \mathsf{then}\, 1 & \mathsf{fix}\, fact(n).\, \mathsf{let}\, r_0 = r_0\, \mathsf{with}\, \{fact = fact, n = n\}\, \mathsf{in} \\
\qquad\qquad\qquad \mathsf{else}\, fact(n-1) \times n]\!]^0 = & \qquad\qquad \mathsf{if}\, r_0{\cdot}n = 0\, \mathsf{then}\, 1 \\
& \qquad\qquad \mathsf{else}\, (r_0{\cdot}fact)(r_0{\cdot}n - 1) \times r_0{\cdot}n
\end{array}
$$

The following example, adapted from [CX03], generates a specialized polynomial calculation function for the polynomial $4+6x+2x^2$; specifically $\lambda x.4+(x\times(6+x\times(2+0)))$. A polynomial is represented as a list of integer values; in this case $[4;6;2]$.

$$
\begin{array}{l}
[\![\mathsf{let}\, poly = \mathsf{fix}\, gen(p).\, \mathsf{if}\, p = \mathtt{nil}\, \mathsf{then}\, \langle 0\rangle \\
\qquad\qquad\qquad\qquad\qquad \mathsf{else}\, \langle\, \grave{}(\mathsf{lift}(\mathsf{hd}\, p)) + x \times\, \grave{}(gen(\mathsf{tl}\, p))\rangle \\
\;\mathsf{in}\, \mathsf{run}\langle \lambda x.\grave{}(poly\, [4;6;2])\rangle\,]\!]^0 =
\end{array}
$$

$$
\begin{aligned}
&\mathsf{let}\, r_0 = r_0\, \mathsf{with}\, \{poly = \mathsf{fix}\, gen(p).\, \mathsf{let}\, r_0 = r_0\, \mathsf{with}\, \{gen = gen, p = p\}\, \mathsf{in} \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{if}\, r_0{\cdot}p = \mathtt{nil}\, \mathsf{then}\, \lambda r_1.0 \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{else}\, \lambda r_1.(\lambda r_2.\mathsf{hd}\ (r_0{\cdot}p))(r_1) + r_1{\cdot}x \times (r_0{\cdot}gen(\mathsf{tl}\, r_0{\cdot}p))(r_1)\} \\
&\mathsf{in}\ (\lambda r_1.\lambda x.\mathsf{let}\, r_1 = r_1\, \mathsf{with}\, \{x = x\}\, \mathsf{in}\, (r_0{\cdot}poly\ [4;6;2])r_1)\{\}
\end{aligned}
$$

Using a record look-up allows for distinguishing variables with the same name in different stages. The examples below illustrate such cases. Note how the occurrence of the variable y in stage 0 is separated from the occurrence in stage 1.

$$[\![\lambda y.\langle y + \text{`}(y)\rangle]\!]^0 = \lambda y.\mathsf{let}\, r_0 = r_0\, \mathsf{with}\, \{y = y\}\, \mathsf{in}\, (\lambda r_1.r_1{\cdot}y + (r_0{\cdot}y)(r_1))$$

$$[\![\lambda y.\langle \lambda y.\text{`}(y)\rangle]\!]^0 = \lambda y.\mathsf{let}\, r_0 = r_0\, \mathsf{with}\, \{y = y\}\, \mathsf{in}\, (\lambda r_1.\lambda y.\mathsf{let}\, r_1 = r_1\, \mathsf{with}\, \{y = y\}\, \mathsf{in}\, (r_0{\cdot}y)(r_1))$$

Being able to translate staged expressions to record calculus expressions brings the question of whether a record type system could be used to type-check staged expressions. This would be desired because record type systems have been studied extensively and have grown mature. The ultimate goal is to use the record type system to decide whether it is safe to execute a staged expression. In particular, we want to be able to say "the staged expression e is type-safe if $[\![e]\!]$ is type-safe." In this chapter we show that this is feasible; the record calculus gives a sound type system for staged computation.

5.2 Type-Checking Program Generators

In this section we give an informal introduction to staged typing. Being one of the state-of-the-art languages for program generation, we take λ^{open}_{poly} [KYC06] as our starting point for a staged type system. We show, by examples, how the type system works, and why an extension with pluggable declarations and subtyping would be desired.

Recall that we take *PG by program construction* as our context. This means that any free variable of a fragment will be captured by its surrounding fragment after filling in a hole, as opposed to *PG by partial evaluation*'s requirement of having a cross-stage binding for every free variable. The λ^{open}_{poly}, being a PG by program construction system, reflects this property in the types it assigns to fragments. A code piece is given a type of the form $\Box(\Gamma \triangleright A)$ with the meaning "the quoted expression will result in a value of type A if it is used in a context that provides the environment Γ." For instance, the fragment $\langle x + 1\rangle$ could be given the type $\Box(\{x : \mathtt{int}\} \triangleright \mathtt{int})$: in an environment that binds x to $\mathtt{int}$, the fragment will result in a value of type $\mathtt{int}$. Row variables [Wan91, Rém94] are used as part of environments for flexibility and quantification. For instance, the above fragment can be typed as $\Box(\{x : \mathtt{int}\}\rho \triangleright \mathtt{int})$, which can be instantiated to types like $\Box(\{x : \mathtt{int}, y : \mathtt{bool}\} \triangleright \mathtt{int})$ or $\Box(\{x : \mathtt{int}, z : \mathtt{int}, w : \mathtt{bool}\} \triangleright \mathtt{int})$ allowing for usage in more contexts.

The function $\lambda c.\langle \mathsf{let}\, x = 1 \,\mathsf{in}\, \grave{}(c)\rangle$ can be typed as $\Box(\{x : \mathtt{int}\}\rho \triangleright \alpha) \to \Box(\rho \triangleright \alpha)^1$ carrying the meaning that the argument of the function has to be a fragment that will receive an environment that maps x to int. If the function is applied on $\langle x+y\rangle$, which could be typed to $\Box(\{x : \mathtt{int}, y : \mathtt{int}\}\rho' \triangleright \mathtt{int})$, the application could be given the type $\Box(\{y : \mathtt{int}\}\rho'' \triangleright \mathtt{int})$. This means that the result of the application should be used in a context that provides an integer value bound to y. Note that the condition for x disappears from the type because the fragment itself binds it to an integer.

We use the operator $\mathsf{run}(\cdot)$ for bringing fragments to stage-0 and executing. Only "complete" program fragments can be run. Therefore, λ^{open}_{poly} allows running a fragment only if it can be given a type $\Box(\varnothing \triangleright A)$, where the empty incoming environment means the fragment does not have any unbound variables. Fragments making other assumptions about outer environments are not runnable.

Library Specialization

In [AK09], we gave a comparative study of several techniques addressing the "library specialization" problem. One of the techniques is program generation. We now use this example to motivate two extensions to the staged type system: pluggable declarations and subtyping.

Very briefly, the problem is this: Given a library with several features, how can we exclude unneeded features so that the library becomes lightweight? The main motivation is to make the library free of unnecessary fields so that its memory fingerprint becomes smaller. Program generation allows customizability of the library by making it possible to include/exclude feature-related code fragments in the library. Take the linked-list class in Figure 5.2 with the "counter" feature that counts the number of operations performed on the list object. To make the counter feature optional, we can write the generator function genLib as shown in the same figure, that takes feature-related code fragments as arguments (cf stands for "counter field", ci stands for "counter increment").

Invoking the genLib function with the arguments ⟨int counter=0;⟩ and ⟨counter++;⟩ would yield a linked-list class with the counter feature included. Passing the empty-code arguments ⟨⟩ and ⟨⟩ produces a linked-list that does not contain the feature, relieving the class from carrying the unneeded field and computation.

[1] Technically row variables are kinded based on their domains [Rém94], and this type is really $\Box(\{x : \mathtt{int}\}\rho_{\{x\}} \triangleright \alpha) \to \Box(\{x : \theta\}\rho_{\{x\}} \triangleright \alpha)$, where $\rho_{\{x\}}$ means that x is not in the domain of ρ, and θ is a "field variable" that stands for a type or the absence of the binding. For brevity, we follow the notational convention used in [KYC06] and denote $\{x : \theta\}\rho$ simply as ρ, and $\rho_{\{x\}}$ as ρ when the full notation can be inferred from the context.

```
class LinkedList {
  Object value;
  LinkedList next;
  int counter=0;

  void add(Object z) {
    counter++;
    ...
  }
  void reverse() {
    counter++;
    ...
  }
  ...
}
```

```
Code genLib(Code cf, Code ci) {
  return ⟨
    class LinkedList {
      Object value;
      LinkedList next;
      `(cf)

      void add(Object z) {
        `(ci)
        ...
      }
      void reverse() {
        `(ci)
        ...
      }
      ...
    } ⟩;
}
```

Figure 5.2: Writing a customizable library using program generation.

Pluggable Declarations

A major motivation of the library specialization problem is to be able to exclude fields. Using program generation to do this requires that the program generation language supports quoting and filling in holes with declarations; we refer to this language feature as "pluggable declarations". In λ_{poly}^{open}, only expressions can be quoted. In Section 5.8 we discuss how it can be extended with pluggable declarations (the extension is actually a syntactic sugaring that could be expressed using existing program generation facilities and higher-order functions).

Assume that λ_{poly}^{open} provides the ability to quote declarations with the syntax $\langle x = e \rangle$, and quoted declarations can be plugged into let-bindings using the syntax $\mathsf{let}\ \grave{}(\cdot)\ \mathsf{in}\, e$. Analogous to a quoted expression, a quoted declaration is given a type $\Diamond(\Gamma_1 \triangleright \Gamma_2)$ with the intuition that "the declaration, when used in a context that provides the environment Γ_1, will output the environment Γ_2." For example, the declaration $\langle x = y+1 \rangle$ can be typed to $\Diamond(\{y : \mathtt{int}\}\rho \triangleright \{y : \mathtt{int}, x : \mathtt{int}\}\rho)$. The function $\lambda d.\langle \mathsf{let}\ \grave{}(d)\ \mathsf{in}\, x + 1 \rangle$ can take the type $\Diamond(\rho_1 \triangleright \{x : \mathtt{int}\}\rho_2) \rightarrow \Box(\rho_1 \triangleright \mathtt{int})$ which carries the meaning that the declaration d should let through an environment that binds x to an $\mathtt{int}$.

Assume also the existence of tuples and arithmetic operations (both would be straightforward extensions). Then, a simple library generator could be abstracted out as below. Suppose for now that the library class contains only one method, v stands for the field of the class, and we return a single function to model the class. Again, cf stands for "counter field" and ci stands for "counter increment".

$$\begin{aligned}\texttt{genLib} = \lambda\texttt{cf}.\ \lambda\texttt{ci}.\ \langle &\mathsf{let}\ \mathtt{v} = 0\ \mathsf{in}\\ &\mathsf{let}\ \grave{}(\texttt{cf})\ \mathsf{in}\\ &(\lambda\mathtt{z}.\grave{}(\texttt{ci})...\mathtt{v} + \mathtt{z})\rangle\end{aligned}$$

We can give the following type to the genLib function above:

$$\forall\rho_1,\rho_2,\gamma.\ \Diamond(\{\mathtt{v}:\mathtt{int}\}\rho_1 \triangleright \{\mathtt{v}:\mathtt{int}\}\rho_2) \to \Box(\{\mathtt{v}:\mathtt{int},\mathtt{z}:\mathtt{int}\}\rho_2 \triangleright \gamma) \to \Box(\rho_1 \triangleright \mathtt{int} \to \mathtt{int})$$

The return type of the generator is $\Box(\rho_1 \triangleright \mathtt{int} \to \mathtt{int})$, which names the outermost environment of the returned code fragment as ρ_1. Then, inside the quoted fragment, a binding for v is made. Hence, the incoming environment of cf is $\{\mathtt{v} : \mathtt{int}\}\rho_1$. Because of the lambda abstraction, a binding for z will be added to the environment that comes out of cf before it goes into ci. Also, this environment has to contain a binding v:int because v is being used as an integer in the body of the method. The type of genLib encapsulates all this information.

The function genLib can be applied on appropriate arguments to generate the desired code. One such application[2] is

$$\texttt{genLib}\ \langle\texttt{cnt} = \mathsf{ref}\ 0\rangle\ \langle\texttt{cnt}{:=}!\texttt{cnt} + 1\rangle$$

Substituting ρ_1 with $\varnothing$, ρ_2 with $\{\texttt{cnt} : \mathtt{int}\ \mathsf{ref}\ \}$, and γ with int gives the type $\Diamond(\{\mathtt{v} : \mathtt{int}\} \triangleright \{\mathtt{v} : \mathtt{int}, \texttt{cnt} : \mathtt{int}\ \mathsf{ref}\ \})$ for the first parameter, and $\Box(\{\mathtt{v} : \mathtt{int}, \mathtt{z} : \mathtt{int}, \texttt{cnt} : \mathtt{int}\ \mathsf{ref}\ \} \triangleright \mathtt{int})$ for the second parameter of the function. Note that the arguments $\langle\texttt{cnt} = \mathsf{ref}\ 0\rangle$ and $\langle\texttt{cnt}{:=}!\texttt{cnt} + 1\rangle$, respectively, have these types as well, making the application legitimate. The result then has type $\Box(\varnothing \triangleright \mathtt{int} \to \mathtt{int})$, which is runnable.

Another possible application of the generator is $\texttt{genLib}\ \langle\rangle\ \langle 0\rangle$, which stands for the case when the feature is excluded. (The second argument is $\langle 0\rangle$ instead of $\langle\rangle$ because we need to provide an expression to the generator function.) This application results in another runnable code value with the same type $\Box(\varnothing \triangleright \mathtt{int} \to \mathtt{int})$. If the generator is applied as $\texttt{genLib}\ \langle\rangle\ \langle\texttt{cnt}{:=}!\texttt{cnt} + 1\rangle$ the type is $\Box(\{\texttt{cnt} : \mathtt{int}\ \mathsf{ref}\ \}\rho \triangleright \mathtt{int} \to \mathtt{int})$. Since the input environment of this type is not empty, the type system does not allow evaluation of the code value via $\mathsf{run}(\cdot)$.

Subtyping

We now slightly modify the library generator example to illustrate the need for the second extension: subtyping. Suppose now the library that will be generated contains two base methods, and hence two uses of ci.

[2]This example contains updatable references. We do not include references initially in the formal presentation. References are added later in Section 5.9.

$$\begin{aligned}\texttt{genLib} = \lambda\texttt{cf}.\ \lambda\texttt{ci}.\ \langle & \mathsf{let}\ \mathtt{v} = 0\ \mathsf{in} \\ & \mathsf{let}\ \grave{}(\mathtt{cf})\ \mathsf{in} \\ & (\lambda\mathtt{z}.\ \grave{}(\mathtt{ci})...\mathtt{v} + \mathtt{z}), \\ & (\lambda\mathtt{y}.\ \grave{}(\mathtt{ci})...\mathtt{y} \times \mathtt{v})\rangle\end{aligned}$$

The environment that goes into the first antiquoted ci is $\{\mathtt{v} : \mathtt{int}, \mathtt{z} : \mathtt{int}\}\rho_2$. However, the incoming environment of the second use is $\{\mathtt{v} : \mathtt{int}, \mathtt{y} : \mathtt{int}\}\rho_2$. Had we the chance to give a polymorphic type to ci such as $\forall\rho.\Box(\{\mathtt{cnt} : \mathtt{int}\ \mathsf{ref}\ \}\rho \rhd \mathtt{int})$, we could instantiate appropriately for the two different uses above. However, because ci is an argument of the generator function, it cannot be used polymorphically (unless we enter the dangerous waters of undecidability and use higher-rank polymorphism); every occurrence of ci has to assume the exact same type. Since the type system has to be conservative because of the possibility that ci may include y or z as a free variable, the derived type for ci says that *both* z and y exist in the incoming environment of ci. So the type given to the generator is

$$\begin{aligned}\forall\rho_1, \rho_2, \gamma.\ \Diamond(\{\mathtt{v} : \mathtt{int}\}\rho_1 \rhd \{\mathtt{v} : \mathtt{int}, \mathtt{y} : \mathtt{int}, \mathtt{z} : \mathtt{int}\}\rho_2) \to \Box(\{\mathtt{v} : \mathtt{int}, \mathtt{y} : \mathtt{int}, \mathtt{z} : \mathtt{int}\}\rho_2 \rhd \gamma) \\ \to \Box(\rho_1 \rhd (\mathtt{int} \to \mathtt{int}) * (\mathtt{int} \to \mathtt{int}))\end{aligned}$$

In this type, the incoming environments of both uses of ci are $\{\mathtt{v} : \mathtt{int}, \mathtt{y} : \mathtt{int}, \mathtt{z} : \mathtt{int}\}\rho_2$, which match exactly the expected environment of ci, as would be required by the type system. However, now an application of the generator that used to be runnable (e.g. $\mathtt{genLib}\ \langle\rangle\ \langle 0\rangle$) gets the type $\Box(\{\mathtt{y} : \mathtt{int}, \mathtt{z} : \mathtt{int}\}\rho \rhd ...)$ — not a runnable type.

Subtyping can get around this problem. We do not have to feed a code fragment with the exact environment that it expects. We can provide a richer environment that still satisfies the fragment's expectations. Take $\langle x + 1\rangle$ with the type $\Box(\{x : \mathtt{int}\} \rhd \mathtt{int})$. It is safe to use this fragment in the environment $\{x : \mathtt{int}\}$, or in $\{x : \mathtt{int}, y : \mathtt{bool}\}$, or in $\{x : \mathtt{int}, w : \mathtt{bool}, k : \mathtt{int} \to \mathtt{int}\}$. As long as the environment provides $\{x : \mathtt{int}\}$ we are fine. This is where subtyping comes into play. Recall that for the above example, the environment that goes into the first antiquoted ci is $\{\mathtt{v} : \mathtt{int}, \mathtt{z} : \mathtt{int}, \mathtt{y} : \theta_1\}\rho_2$, and the second use is $\{\mathtt{v} : \mathtt{int}, \mathtt{z} : \theta_2, \mathtt{y} : \mathtt{int}\}\rho_2$. If we give ci the type $\{\mathtt{v} : \mathtt{int}\}\rho_2$, using the properties

$$\{\mathtt{v} : \mathtt{int}, \mathtt{z} : \mathtt{int}, \mathtt{y} : \theta_1\}\rho_2 <: \{\mathtt{v} : \mathtt{int}\}\rho_2$$

$$\{\mathtt{v} : \mathtt{int}, \mathtt{z} : \theta_2, \mathtt{y} : \mathtt{int}\}\rho_2 <: \{\mathtt{v} : \mathtt{int}\}\rho_2$$

we can successfully obtain a runnable type as expected. We give in Section 5.7 more details of subtyping and show how Pottier's subtyping constraints [Pot00b] can be used to come up with a staged type system with subtyping that solves the problem of having superfluous requirements on an incoming environment.

$$
\begin{aligned}
&x \in Var\\
&c \in Constant\\
&e \in Exp ::= c \mid x \mid \lambda x.e \mid \lambda^* x.e \mid \mathsf{fix}\, f(x).\, e \mid e\, e \mid \mathsf{let}\, x = e \,\mathsf{in}\, e\\
&\qquad\qquad \mid \langle e \rangle \mid \grave{}(e) \mid \mathsf{run}(e) \mid \mathsf{lift}(e)
\end{aligned}
$$

Figure 5.3: Syntax of λ^{gen}_{poly}.

$$
\begin{array}{r|r}
\begin{aligned}
FV^n(c) &= \varnothing\\
FV^0(x) &= \{x\}\\
FV^{n+1}(x) &= \varnothing\\
FV^0(\lambda x.e) &= FV^0(e) \setminus \{x\}\\
FV^{n+1}(\lambda x.e) &= FV^{n+1}(e)\\
FV^n(\lambda^* x.e) &= FV^n(\lambda x.e)\\
FV^n(\langle e \rangle) &= FV^{n+1}(e)\\
FV^{n+1}(\grave{}(e)) &= FV^n(e)
\end{aligned}
&
\begin{aligned}
FV^n(e_1 e_2) &= FV^n(e_1) \cup FV^n(e_2)\\
FV^0(\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2) &= FV^0(e_1) \cup (FV^0(e_2) \setminus \{x\})\\
FV^{n+1}(\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2) &= FV^{n+1}(e_1) \cup FV^{n+1}(e_2)\\
FV^0(\mathsf{fix}\, f(x).\, e) &= FV^0(e) \setminus \{f, x\}\\
FV^{n+1}(\mathsf{fix}\, f(x).\, e) &= FV^{n+1}(e)\\
FV^n(\mathsf{run}(e)) &= FV^n(e)\\
FV^n(\mathsf{lift}(e)) &= FV^n(e)
\end{aligned}
\end{array}
$$

Figure 5.4: Finding the stage-0 free variables of λ^{gen}_{poly} expression.

5.3 Staged Language

In this section we give the formal definition of the staged language we use. The language is defined based on λ^{open}_{poly}[KYC06] — an ML-like language that supports program generation with freely-open fragments, references, let-polymorphism, and variable hygiene. For the moment we exclude references and open[3], but add a fix-point operator to have recursion, and call this language with core program generation facilities λ^{gen}_{poly}. The syntax of the language is given in Figure 5.3. λ^* is hygienic variable binding; it uniquely renames the bound variable to avoid capturing a variable after a fragment in plugged in the scope of the binding. $\mathsf{lift}(\cdot)$ raises a value to the next stage. Extension of the language with references and pluggable declarations is discussed later.

We use the syntax $\langle \cdot \rangle$ for quotation (box in λ^{open}_{poly}), and $\grave{}(\cdot)$ for antiquotation (unbox_1 in λ^{open}_{poly}). A quotation denotes a computation in the next stage whereas an antiquotation denotes a computation in the previous stage. There is no multi-stage antiquotation like λ^{open}_{poly}'s unbox_k. This can be achieved by nesting antiquotations k times. We use $\mathsf{run}(\cdot)$, instead of λ^{open}_{poly}'s unbox_0 to evaluate code values.

5.3.1 Auxiliary Definitions

In this section we give auxiliary definitions that are used in the operational semantics and the type system of the staged calculus.

Definition 5.3.1. The domain of a function f is denoted as $dom(f)$.

[3] open is a syntax-directed subtyping operator restricted to closed fragments. Our extension with subtyping, discussed later in the chapter, subsumes open.

$$
\begin{array}{rcl}
c[x\backslash e]^n & = & c \\
x[x\backslash e]^0 & = & e \\
y[x\backslash e]^0 & = & y\text{, if } y \neq x \\
y[x\backslash e]^{n+1} & = & y \\
(\lambda x.e)[x\backslash e']^0 & = & \lambda x.e \\
(\lambda y.e)[x\backslash e']^0 & = & \lambda z.e[y\backslash z]^0[x\backslash e']^0 \\
 & & \qquad \text{where } z \text{ is fresh and } y \neq x \\
(\lambda y.e)[x\backslash e']^{n+1} & = & \lambda y.e[x\backslash e']^{n+1} \\
(\lambda^* x.e)[x\backslash e']^0 & = & \lambda^* x.e \\
(\lambda^* y.e)[x\backslash e']^0 & = & \lambda^* z.e[y\backslash z]^0[x\backslash e']^0 \\
 & & \qquad \text{where } z \text{ is fresh and } y \neq x \\
(\lambda^* y.e)[x\backslash e']^{n+1} & = & \lambda^* y.e[x\backslash e']^{n+1} \\
(\mathsf{fix}\, f(y).\, e)[x\backslash e']^n & = & \textit{similar to abstraction} \\
(e_1 e_2)[x\backslash e]^n & = & e_1[x\backslash e]^n\ e_2[x\backslash e]^n \\
(\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2)[x\backslash e]^0 & = & \mathsf{let}\, x = e_1[x\backslash e]^0\, \mathsf{in}\, e_2 \\
(\mathsf{let}\, y = e_1\, \mathsf{in}\, e_2)[x\backslash e]^0 & = & \mathsf{let}\, z = e_1[x\backslash e]^0\, \mathsf{in}\, e_2[y\backslash z]^0[x\backslash e]^0 \\
 & & \qquad \text{where } z \text{ is fresh and } y \neq x \\
(\mathsf{let}\, y = e_1\, \mathsf{in}\, e_2)[x\backslash e]^{n+1} & = & \mathsf{let}\, y = e_1[x\backslash e]^{n+1}\, \mathsf{in}\, e_2[x\backslash e]^{n+1} \\
\langle e\rangle[x\backslash e']^n & = & \langle e[x\backslash e']^{n+1}\rangle \\
\grave{}(e)[x\backslash e']^{n+1} & = & \grave{}(e[x\backslash e']^n) \\
\mathsf{run}(e)[x\backslash e']^n & = & \mathsf{run}(e[x\backslash e']^n) \\
\mathsf{lift}(e)[x\backslash e']^n & = & \mathsf{lift}(e[x\backslash e']^n)
\end{array}
$$

Figure 5.5: Staged substitution.

Definition 5.3.2. The function update operator, $\triangleleft\!\!+$, is defined as follows:

$$
(f \triangleleft\!\!+ g)(x) = \begin{cases} g(x), & \text{if } x \in dom(g) \\ f(x), & \text{otherwise} \end{cases}
$$

Definition 5.3.3. The *depth* of an expression e is the maximum number of nested antiquotations in e that are not enclosed by quotations.

Definition 5.3.4. An expression e is a *stage-n expression* if the depth of e is less than or equal to n. This also means that a stage-n expression is also a stage-$(n + 1)$ expression.

Definition 5.3.5. The free variables of a staged expression are the free variables at stage 0. The definition is in Figure 5.4.

Definition 5.3.6. Substitution in a staged expression replaces variables at stage-0 and is defined in Figure 5.5.

Definition 5.3.7. Staged renaming $[x^n \overset{m}{\mapsto} z]e$ replaces with z the occurrences of the stage-n variable x in the stage-m expression e. Staged renaming is used in operations regarding λ^*. The definition of staged renaming is in [KYC06].

$$
\begin{array}{l}
v^n \in Val^n \\
Val^0 \quad ::= c \mid \lambda x.e \mid \mathsf{fix}\, f(x).\, e \mid \langle v^1 \rangle \\
Val^{n+1} ::= c \mid x \mid \lambda x.v^{n+1} \mid \mathsf{fix}\, f(x).\, v^{n+1} \mid v^{n+1} v^{n+1} \\
\qquad\qquad \mid \langle v^{n+2} \rangle \mid \mathsf{lift}(v^{n+1}) \mid \mathsf{run}(v^{n+1}) \mid \mathsf{let}\, x = v^{n+1}\, \mathsf{in}\, v^{n+1} \\
\qquad\qquad \mid \text{\`{}}(v^n) \quad \text{(if } n > 0\text{)}
\end{array}
$$

Figure 5.6: The definition of values in λ^{gen}_{poly}.

ESABS
$$\frac{e \longrightarrow_{n+1} e'}{\lambda x.e \longrightarrow_{n+1} \lambda x.e'}$$

ESSYM
$$\frac{z \text{ is fresh for } e}{\lambda^* x.e \longrightarrow_n \lambda z.[x^n \overset{n}{\mapsto} z]e}$$

ESFIX
$$\frac{e \longrightarrow_{n+1} e'}{\mathsf{fix}\, f(x).\, e \longrightarrow_{n+1} \mathsf{fix}\, f(x).\, e'}$$

ESAPP
$$\frac{e_1 \longrightarrow_n e_1'}{e_1 e_2 \longrightarrow_n e_1' e_2} \qquad \frac{e_1 \in Val^n \quad e_2 \longrightarrow_n e_2'}{e_1 e_2 \longrightarrow_n e_1 e_2'} \qquad \frac{e_2 \in Val^0}{(\lambda x.e) e_2 \longrightarrow_0 e[x \backslash e_2]^0}$$

$$\frac{e_2 \in Val^0}{(\mathsf{fix}\, f(x).\, e) e_2 \longrightarrow_0 e[f \backslash \mathsf{fix}\, f(x).\, e]^0 [x \backslash e_2]^0}$$

ESLET
$$\frac{e_1 \longrightarrow_n e_1'}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_n \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_2} \qquad \frac{e_1 \in Val^0}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_0 e_2[x \backslash e_1]^0}$$

$$\frac{e_1 \in Val^{n+1} \quad e_2 \longrightarrow_{n+1} e_2'}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \longrightarrow_{n+1} \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2'}$$

ESBOX
$$\frac{e \longrightarrow_{n+1} e'}{\langle e \rangle \longrightarrow_n \langle e' \rangle}$$

ESUBOX
$$\frac{e \longrightarrow_n e'}{\text{\`{}}(e) \longrightarrow_{n+1} \text{\`{}}(e')} \qquad \frac{e \in Val^1}{\text{\`{}}(\langle e \rangle) \longrightarrow_1 e}$$

ESRUN
$$\frac{e \longrightarrow_n e'}{\mathsf{run}(e) \longrightarrow_n \mathsf{run}(e')} \qquad \frac{e \in Val^1}{\mathsf{run}(\langle e \rangle) \longrightarrow_0 e}$$

ESLIFT
$$\frac{e \longrightarrow_n e'}{\mathsf{lift}(e) \longrightarrow_n \mathsf{lift}(e')} \qquad \frac{e \in Val^0}{\mathsf{lift}(e) \longrightarrow_0 \langle e \rangle}$$

Figure 5.7: The small-step semantics of λ^{gen}_{poly}.

5.3.2 Operational Semantics

Despite the large number of different program generation languages, their dynamic semantics for the core program generation constructs are almost the same. Definitions of operational semantics can be found in [CMT04, DP96, KKcS08, KYC06, MTBS99, Rhi05]. A big-step operational semantics of λ^{open}_{poly} is given in [KYC06]. We choose to present a small-step semantics of λ^{gen}_{poly}, adapted from [Rhi05]. Having added recursion to the language, small-step semantics serves better when reasoning about non-terminating reductions. The values are given in Figure 5.6 and the reduction rules in Figure 5.7. A reduction takes place at a certain stage. A quotation increments the stage while an antiquotation decrements it. The reduction of an expression at a stage higher than 0 recurses into the subexpressions because the subexpressions may contain holes that bring the reduction to stage 0, or that may be filled in at stage 1. The second rule of ESUBOX is where the actual hole-filling occurs. Note that this rule is defined specifically for stage 1; for optimization purposes, we could define it to take place at any stage $n > 0$, however, this would make the application of the rules non-deterministic. So, this ESUBOX rule is only defined at stage 1.

5.3.3 Type System

We give the λ^{gen}_{poly} type system, adapted from λ^{open}_{poly}[KYC06]. The definition of types is given in Figure 5.8; typing rules are in Figure 5.9. A type can be one of (i) a type variable α, (ii) a constant type ι, (iii) a function type $A \to B$, or (iv) a box-type $\Box(\Gamma \triangleright A)$. Code values type to box-types. A box-type $\Box(\Gamma \triangleright A)$ has the meaning "the fragment will result in a value of type A if it is evaluated in a context that provides the environment Γ." The fields in an environment Γ can be one of (i) a type A, (ii) `Abs`, denoting the absence of the binding for that particular field, or, (iii) a field variable θ. In a judgment $\Delta_0, \ldots, \Delta_n \vdash_S e : A$, the typing environment Δ_i stands for the environment of stage i. Quotations and antiquotations add or remove new typing environments.

Definition 5.3.8. A type scheme $\forall\psi.\sigma$ binds the variable ψ in the standard way. FV returns the set of free variables in a type (scheme).

Definition 5.3.9. For convenience, we denote $\forall\psi_1.\forall\psi_2.\ldots\forall\psi_n.A$ as $\forall\vec{\psi}.A$, where $\vec{\psi}$ stands for $\psi_1 \ldots \psi_n$.

Definition 5.3.10. Generalization of a type to a type scheme with respect to a list of type scheme environments is defined as

$$\mathrm{GEN}_A(\Delta_0, \ldots, \Delta_n) = \forall\vec{\psi}.A \qquad \text{where } \vec{\psi} = FV(A) \setminus FV(\Delta_0, \ldots, \Delta_n)$$

Definition 5.3.11. A substitution φ is a partial function from type system variables to types. We extend the definition of a substitution to apply on a compound object in the obvious way.

$$
\begin{array}{lll}
\alpha, \beta & \in & STyVar \\
A, B & \in & SType ::= \alpha \mid \iota \mid A \to B \mid \Box(\Gamma \triangleright A) \\
\theta & \in & SFieldVar \\
F & \in & SField ::= A \mid \mathtt{Abs} \mid \theta \\
\rho & \in & SEnvVar \\
\Gamma & \in & SEnv = Var \rightharpoonup SField \\
 & & ::= \{x_i : F_i\}_1^m \mid \{x_i : F_i\}_1^m \rho \\
\psi & \in & SEnvVar \oplus SFieldVar \\
\sigma & \in & STyScheme ::= \forall \psi.\sigma \mid A \\
\mu & \in & SFieldScheme ::= \sigma \mid \mathtt{Abs} \\
\Delta & \in & STySchemeEnv = Var \rightharpoonup SFieldScheme \\
 & & ::= \{x_i : \mu_i\}_1^m \mid \{x_i : \mu_i\}_1^m \rho
\end{array}
$$

Figure 5.8: The definition of types in λ^{gen}_{poly}.

TSCON
$$\Delta_0, \ldots, \Delta_n \vdash_S c : \iota$$

TSVAR
$$\frac{A \prec \Delta_n(x)}{\Delta_0, \ldots, \Delta_n \vdash_S x : A}$$

TSABS
$$\frac{\Delta_0, \ldots, \Delta_n \Leftarrow \{x : A\} \vdash_S e : B}{\Delta_0, \ldots, \Delta_n \vdash_S \lambda x.e : A \to B}$$

TSSYM
$$\frac{\Delta_0, \ldots, \Delta_n \vdash_S \lambda z.[x^n \overset{n}{\mapsto} z]e : A \quad z \text{ is fresh for } e}{\Delta_0, \ldots, \Delta_n \vdash_S \lambda^* x.e : A}$$

TSFIX
$$\frac{\Delta_0, \ldots, \Delta_n \Leftarrow \{f : A \to B, x : A\} \vdash_S e : B}{\Delta_0, \ldots, \Delta_n \vdash_S \mathsf{fix}\, f(x).\, e : A \to B}$$

TSAPP
$$\frac{\Delta_0, \ldots, \Delta_n \vdash_S e_1 : A \to B \quad \Delta_0, \ldots, \Delta_n \vdash_S e_2 : A}{\Delta_0, \ldots, \Delta_n \vdash_S e_1 e_2 : B}$$

TSLET
$$\frac{\Delta_0, \ldots, \Delta_n \vdash_S e_1 : A \quad \Delta_0, \ldots, \Delta_n \Leftarrow \{x : \mathrm{GEN}_A(\Delta_0, \cdots, \Delta_n)\} \vdash_S e_2 : B}{\Delta_0, \ldots, \Delta_n \vdash_S \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : B}$$

TSBOX
$$\frac{\Delta_0, \ldots, \Delta_n, \Gamma \vdash_S e : A}{\Delta_0, \ldots, \Delta_n \vdash_S \langle e \rangle : \Box(\Gamma \triangleright A)}$$

TSUNBOX
$$\frac{\Delta_0, \ldots, \Delta_n \vdash_S e : \Box(\Gamma \triangleright A) \quad \Gamma \prec \Delta_{n+1}}{\Delta_0, \ldots, \Delta_n, \Delta_{n+1} \vdash_S \text{`}(e) : A}$$

TSRUN
$$\frac{\Delta_0, \ldots, \Delta_n \vdash_S e : \Box(\varnothing \triangleright A)}{\Delta_0, \ldots, \Delta_n \vdash_S \mathsf{run}(e) : A}$$

TSLIFT
$$\frac{\Delta_0, \ldots, \Delta_n \vdash_S e : A}{\Delta_0, \ldots, \Delta_n \vdash_S \mathsf{lift}(e) : \Box(\Gamma \triangleright A)}$$

Figure 5.9: The λ^{open}_{poly}[KYC06] type system rules adapted for λ^{gen}_{poly}– a language with core program generation facilities, recursion, and no references.

$$
\begin{aligned}
&x \in Var \\
&a \in Label = Var \\
&r \in RVar \\
&w, f \in Name = Var \cup RVar \\
&c \in Constant \\
&e \in RExp ::= c \mid w \mid \lambda w.e \mid \mathsf{fix}\, f(x).\, e \mid e\, e \mid \mathsf{let}\, w = e \,\mathsf{in}\, e \\
&\qquad\qquad\quad \mid \{\} \mid e \,\mathsf{with}\, \{a = e\} \mid e{\cdot}a
\end{aligned}
$$

Figure 5.10: Record calculus syntax.

We assume that all substitutions respect domains of variables. That is, a type variable α is mapped to an $A \in SType$; an environment variable ρ is mapped to a $\Gamma \in SEnv$; and a field variable θ is mapped to an $F \in SField$. Hence, $\varphi A \in SType$ for any A; $\varphi\Gamma \in SEnv$ for any Γ; and $\varphi F \in SField$ for any F.

Definition 5.3.12 (Instantiation)**.** A type A is an instance of a type scheme $\forall\vec{\psi}.A'$, written $A \prec \forall\vec{\psi}.A'$, if and only if there is a substitution φ with domain $\vec{\psi}$ such that $\varphi A' = A$.

A type scheme σ is more general than a type scheme σ', denoted $\sigma' \prec \sigma$ with a slight abuse of notation, if and only if $A \prec \sigma$ for any $A \prec \sigma'$.

Environment instantiation, overloading $\prec$, is defined as follows:

$$
\begin{aligned}
\{x_i : F_i\}_1^m \prec \{x_i : \mu_i\}_1^m \quad &\Longleftrightarrow \quad F_i \prec \mu_i \text{ for any } i \in [1..m] \\
\{x_i : F_i\}_1^m \rho \prec \{x_i : \mu_i\}_1^m \rho \quad &\Longleftrightarrow \quad F_i \prec \mu_i \text{ for any } i \in [1..m]
\end{aligned}
$$

For all the typing rules in Figure 5.9, the difference from λ_{poly}^{open} is that, due to the absence of references, there is no store typing that is being threaded through a proof tree. The TSLET rule, additionally, does not distinguish between expansive and non-expansive expressions. The soundness of this type system with respect to the operational semantics is given in [KYC06].

5.4 Record Language

Let λ_{poly}^{rec} be a record calculus with the exception that the record and non-record variables are disjoint. Records are mappings from labels to values. The syntax of λ_{poly}^{rec} is given in Figure 5.10. The record operations are (1) record update via the with operator, (2) accessing a value in a record using the label, and (3) the empty record.

Definition 5.4.1. We use the shorter notation $\{a_1 = e_1, a_2 = e_2, \ldots, a_m = e_m\}$ for the expression $\{\}\,\mathsf{with}\,\{a_1 = e_1\}\,\mathsf{with}\,\{a_2 = e_2\}\ldots\,\mathsf{with}\,\{a_m = e_m\}$.

5.4.1 Auxiliary Definitions

Definition 5.4.2. The free variables of a record calculus expression are defined as follows.

$$
\begin{array}{rcl|rcl}
FV(c) &=& \varnothing & FV(\mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2) &=& FV(e_1) \cup (FV(e_2) \setminus \{w\}) \\
FV(w) &=& \{w\} & FV(\{\}) &=& \varnothing \\
FV(\lambda w.e) &=& FV(e) \setminus \{w\} & FV(e_1 \,\mathsf{with}\, \{a = e_2\}) &=& FV(e_1) \cup FV(e_2) \\
FV(\mathsf{fix}\, f(x).\, e) &=& FV(e) \setminus \{f, x\} & FV(e \cdot a) &=& FV(e) \\
FV(e_1 e_2) &=& FV(e_1) \cup FV(e_2) & & &
\end{array}
$$

5.4.2 Operational Semantics

We give operational semantics of the record calculus using unrestricted reductions, where simplifications can be performed under lambda abstractions as well. We will give call-by-value semantics when we introduce updatable references into the language (in Section 5.9). The reductions are performed according to the following rules.

$$
\begin{array}{rcl}
(\lambda w.e_1)e_2 & \longrightarrow_\beta & e_1[w \backslash e_2] \\
\mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2 & \longrightarrow_\beta & e_2[w \backslash e_1] \\
(e_2 \,\mathsf{with}\, \{a_1 = e_1\}) \cdot a_2 & \longrightarrow_\beta & e_2 \cdot a_2 \text{ if } a_1 \neq a_2 \\
(e_2 \,\mathsf{with}\, \{a = e_1\}) \cdot a & \longrightarrow_\beta & e_1 \\
e \,\mathsf{with}\, \{a_1 = e_1\} \,\mathsf{with}\, \{a_2 = e_2\} & \longrightarrow_\beta & e \,\mathsf{with}\, \{a_2 = e_2\} \,\mathsf{with}\, \{a_1 = e_1\} \text{ if } a_1 \neq a_2 \\
e \,\mathsf{with}\, \{a = e_1\} \,\mathsf{with}\, \{a = e_2\} & \longrightarrow_\beta & e \,\mathsf{with}\, \{a = e_2\}
\end{array}
$$

A reduction is the congruence closure of the rules above. That is, if an expression e_1 is inside a context $C[\,]$ and $e_1 \longrightarrow_\beta e_2$, then $C[e_1] \longrightarrow_\beta C[e_2]$.

5.4.3 Type System

The definition of types and other objects is in Figure 5.11; typing rules are in Figure 5.12. This record type system is not completely standard. One can notice that (1) we distinguish between record variables and non-record variables, (2) the grammar of types does not allow construction of certain types that would normally be allowed in a standard record calculus; in particular, we want to avoid having types of the form $T \rightarrow \Gamma$. These changes are needed to make the type system sound with respect to the staged semantics. Although this type system is more restricted than a standard record type system (i.e. we cannot type-check as many expressions), it is still sound with respect to the record semantics, and we do not lose expressiveness with respect to the staged semantics. (We will see that we obtain a type system equal to λ^{gen}_{poly}.) The essence of the need for these changes in the definition of the types comes from the fact that a quoted expression is translated to a function.

$$
\begin{array}{lcl}
\alpha, \beta & \in & RLegTyVar \\
A, B & \in & RLegType ::= \alpha \mid \iota \mid T \to A \\
T & \in & RType ::= A \mid \Gamma \\
\theta & \in & RFieldVar \\
F & \in & RField ::= A \mid \mathtt{Abs} \mid \theta \\
\rho & \in & RRecVar \\
\Gamma & \in & RRec = Label \rightharpoonup RField \\
 & & ::= \{a_i : F_i\}_1^m \mid \{a_i : F_i\}_1^m \rho \\
\psi & \in & RTyVar \oplus RRecVar \oplus RFieldVar \\
\sigma & \in & RTyScheme ::= \forall \psi.\sigma \mid T \\
\Delta & \in & RTySchemeEnv = Name \rightharpoonup RTyScheme
\end{array}
$$

Figure 5.11: The definition of types in the record calculus.

TRCON $$\Delta \vdash_R c : \iota$$

TRVAR $$\frac{A \prec \Delta(x)}{\Delta \vdash_R x : A} \qquad \frac{\Gamma \prec \Delta(r)}{\Delta \vdash_R r : \Gamma}$$

TRABS $$\frac{\Delta \Leftarrow \{x : A\} \vdash_R e : B}{\Delta \vdash_R \lambda x.e : A \to B} \qquad \frac{\Delta \Leftarrow \{r : \Gamma\} \vdash_R e : B}{\Delta \vdash_R \lambda r.e : \Gamma \to B}$$

TRFIX $$\frac{\Delta \Leftarrow \{f : A \to B, x : A\} \vdash_R e : B}{\Delta \vdash_R \mathsf{fix}\, f(x).\, e : A \to B}$$

TRAPP $$\frac{\Delta \vdash_R e_1 : T \to B \quad \Delta \vdash_R e_2 : T}{\Delta \vdash_R e_1 e_2 : B}$$

TRLET $$\frac{\Delta \vdash_R e_1 : A \quad \Delta \Leftarrow \{x : \mathrm{GEN}_A(\Delta)\} \vdash_R e_2 : B}{\Delta \vdash_R \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : B} \qquad \frac{\Delta \vdash_R e_1 : \Gamma \quad \Delta \Leftarrow \{r : \mathrm{GEN}_\Gamma(\Delta)\} \vdash_R e_2 : B}{\Delta \vdash_R \mathsf{let}\, r = e_1 \,\mathsf{in}\, e_2 : B}$$

TRACC $$\frac{\Delta \vdash_R e : \Gamma \quad \Gamma(a) = A}{\Delta \vdash_R e{\cdot}a : A}$$

TREMPTY $$\Delta \vdash_R \{\} : \varnothing$$

TRUPD $$\frac{\Delta \vdash_R e_1 : \Gamma \quad \Delta \vdash_R e_2 : A}{\Delta \vdash_R e_1 \,\mathsf{with}\, \{a = e_2\} : \Gamma \Leftarrow \{a : A\}}$$

Figure 5.12: The type system of the record calculus.

Let us now illustrate with examples how these restrictions help:

- **Why should record variables be separated from non-record variables?**
  Consider the expression $\langle 42 \rangle\, 2$. This is ill-typed because a quoted expression is being used as a function. The translation of this expression at stage 0 is $(\lambda r_1.42)2$. If we do not distinguish record variables, the lambda abstraction in the translation can be given the type `int` $\to$ `int`, meaning that $(\lambda r_1.42)2$ would pass the type-checker. This is certainly not wanted. Restricting record variables to record types prevents this kind of failure.

  For similar reasons, non-record variables should be restricted to non-record types. An ill-typed staged expression whose translation would otherwise pass the record type system is $\lambda x.\langle 42 \rangle\, x$. Assigning a record type to the variable x would yield a valid type if non-record variables are not restricted to non-record types.

- **Why should types in the form $T \to \Gamma$ not be allowed?**
  Restricting non-record variables to non-record types is not sufficient. A quoted expression is translated to a function, but we want to apply this function only when filling in a hole or running the expression. Other applications should not be allowed. If types of the form $T \to \Gamma$ could be constructed, we could create an expression which results in the type Γ, and could feed this type to the lambda abstraction that represents the quoted expression. Consider the expression $\lambda x.\lambda y.\langle 42\rangle\,(x\;y)$. Assigning the type $\mathtt{int} \to \varnothing$ to x and $\mathtt{int}$ to y would yield a valid type for the translation of this expression, which is ill-typed in the staged semantics.

The record type system enjoys the following (standard) definitions and lemmas.

Definition 5.4.3.

$$\mathrm{GEN}_T(\Delta) = \forall\vec{\psi}.T \text{ where } \vec{\psi} = FV(T) \setminus FV(\Delta)$$

Definition 5.4.4. A substitution φ is a partial function from type system variables to types. We extend the definition of a substitution to apply on a compound object in the obvious way.

We assume that all substitutions respect domains of variables. That is, a type variable α is mapped to an $A \in RLegType$; a record variable ρ is mapped to a $\Gamma \in RRec$; and a field variable θ is mapped to an $F \in RField$. Hence, $\varphi A \in RLegType$ for any A; $\varphi\Gamma \in RRec$ for any Γ; and $\varphi F \in RField$ for any F.

Definition 5.4.5 (Instantiation)**.** A type T is an instance of a type scheme $\forall\vec{\psi}.T'$, written $T \prec \forall\vec{\psi}.T'$, if and only if there is a substitution φ with domain $\vec{\psi}$ such that $\varphi T' = T$.

A type scheme σ is more general than a type scheme σ', denoted $\sigma' \prec \sigma$ with a slight abuse of the notation, if and only if $T \prec \sigma$ for any $T \prec \sigma'$.

The record calculus satisfies the standard lemmas such as Weakening/Strengthening, Substitution, Generalization, Preservation, and Progress. Standard proofs in the style of Wright and Felleisen [WF94] apply with minor changes.

5.5 Transformation

In this section we provide the definition of a translation, $[\![\cdot]\!]_{R_0,\ldots,R_n}$, from λ^{gen}_{poly} expressions to λ^{rec}_{poly}. This is an improved version of the initial translation given in Figure 5.1. We do not use that original translation because the improved version provides a more useful result about the relation between staged and record operational semantics as well as making some of the proofs less complicated. Consider the expression $\langle\lambda x.x + y\rangle$. This would be translated by the first translation to $(\lambda r_1.\lambda x.\mathsf{let}\, r_1 = r_1 \,\mathsf{with}\, \{x = x\} \,\mathsf{in}\, r_1 \cdot x + r_1 \cdot y)$. Note that the binding of x is in the fragment. Therefore there is no need to access x through a record; we could as well translate $\langle \lambda x.x + y \rangle$ to $(\lambda r_1.\lambda x.x + r_1 \cdot y)$, where only the free variable y is looked up in the environment. The new translation does exactly that: If a variable already exists in the scope, no record lookup for that variable is made. However, we need to be careful about variables with the same name that occur in different stages, because when the quotations are removed, a higher-stage binding may capture a lower stage variable. Take the expression $(\lambda y.\langle \lambda y.\,\grave{}(y) + y \rangle)$. It is wrong to simply translate it to $(\lambda y.\lambda r_1.\lambda y.y(r_1) + y)$. It should rather be translated to $(\lambda z.\lambda r_1.\lambda w.z(r_1) + w)$, which preserves the meaning. For this purpose, we use "renaming environments", denoted by the subscript $R_0, \ldots, R_n$, where R_i carries the variables bound so far at stage i. It maps them to fresh names so that we can replace a variable avoiding any unintentional capture. The translation is given in Figure 5.13. The definition of a renaming environment is below.

Definition 5.5.1 (Renaming environment)**.** A renaming environment R is defined as follows.

$$R \in \mathit{RenamingEnv} ::= \{\} \mid r \mid R\,\mathsf{with}\,\{x = y\}$$

A renaming environment defines a function from variables to record expressions:

$$\begin{array}{rcl}
(R\,\mathsf{with}\,\{x = y\})(x) & = & y \\
(R\,\mathsf{with}\,\{z = y\})(x) & = & R(x) \text{ if } x \neq z \\
r(x) & = & r \cdot x \\
\{\}(x) & = & \textbf{error}
\end{array}$$

The domain of a renaming environment is the set of variables for which there are explicit mappings:

$$\begin{array}{rcl}
\mathit{dom}(R\,\mathsf{with}\,\{x = y\}) & = & \mathit{dom}(R) \cup \{x\} \\
\mathit{dom}(r) & = & \{\} \\
\mathit{dom}(\{\}) & = & \{\}
\end{array}$$

Throughout this chapter, we assume that free variables in a renaming environment are unique. That is, for any renaming environment sequence $R_0, \ldots, R_n$ in $[\![e]\!]_{R_0,\ldots,R_n}$ we have

(i) $z \notin FV(R'_i) \cup FV^n(e)$ for any $R_i = R'_i\,\mathsf{with}\,\{x = z\}$

(ii) $FV(R_i) \cap FV(R_j) = \varnothing$ if $i \neq j$

Note that these conditions are preserved by the transformation. Also, in order to reduce notational clutter, we assume that the record variable in R_n is r_n.

$$\llbracket c \rrbracket_{R_0,\ldots,R_n} = c$$
$$\llbracket x \rrbracket_{R_0,\ldots,R_n} = R_n(x)$$
$$\llbracket \lambda x.e \rrbracket_{R_0,\ldots,R_n} = \lambda z.\ \llbracket e \rrbracket_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} \text{ where } z \text{ is fresh}$$
$$\llbracket \lambda^* x.e \rrbracket_{R_0,\ldots,R_n} = \llbracket \lambda z.[x^n \overset{n}{\mapsto} z]e \rrbracket_{R_0,\ldots,R_n} \text{ where } z \text{ is fresh}$$
$$\llbracket \mathsf{fix}\, f(x).\ e \rrbracket_{R_0,\ldots,R_n} = \mathsf{fix}\, g(z).\ \llbracket e \rrbracket_{R_0,\ldots,R_n \,\mathsf{with}\, \{f=g,x=z\}} \text{ where } g, z \text{ are fresh}$$
$$\llbracket e_1 e_2 \rrbracket_{R_0,\ldots,R_n} = \llbracket e_1 \rrbracket_{R_0,\ldots,R_n} \llbracket e_2 \rrbracket_{R_0,\ldots,R_n}$$
$$\llbracket \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \rrbracket_{R_0,\ldots,R_n} = \mathsf{let}\, z = \llbracket e_1 \rrbracket_{R_0,\ldots,R_n} \,\mathsf{in}\, \llbracket e_2 \rrbracket_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} \text{ where } z \text{ is fresh}$$
$$\llbracket \langle e \rangle \rrbracket_{R_0,\ldots,R_n} = \lambda r.\ \llbracket e \rrbracket_{R_0,\ldots,R_n,r} \text{ where } r \text{ is fresh}$$
$$\llbracket \,\grave{}\,(e) \rrbracket_{R_0,\ldots,R_n,R_{n+1}} = (\llbracket e \rrbracket_{R_0,\ldots,R_n})\ R_{n+1}$$
$$\llbracket \mathsf{run}(e) \rrbracket_{R_0,\ldots,R_n} = (\llbracket e \rrbracket_{R_0,\ldots,R_n})\{\}$$
$$\llbracket \mathsf{lift}(e) \rrbracket_{R_0,\ldots,R_n} = \lambda r.\ \llbracket e \rrbracket_{R_0,\ldots,R_n} \text{ where } r \text{ is fresh}$$

Figure 5.13: Transformation from λ^{gen}_{poly} expressions to λ^{rec}_{poly}.

$$core(\mathtt{Abs}) = \mathtt{Abs}$$
$$core(\forall\vec{\psi}.A) = A$$
$$\llbracket \iota \rrbracket = \iota$$
$$\llbracket \psi \rrbracket = \psi$$
$$\llbracket A \to B \rrbracket = \llbracket A \rrbracket \to \llbracket B \rrbracket$$
$$\llbracket \Box(\Gamma \triangleright A) \rrbracket = \llbracket \Gamma \rrbracket \to \llbracket A \rrbracket$$
$$\llbracket \mathtt{Abs} \rrbracket = \mathtt{Abs}$$
$$\llbracket \forall\psi.\sigma \rrbracket = \forall\psi.\llbracket \sigma \rrbracket$$
$$\llbracket \{x_1 : \mu_1, \ldots, x_m : \mu_m\}\rho \rrbracket = \forall\vec{\psi}.\{x_1 : \llbracket core(\mu_1) \rrbracket, \ldots, x_m : \llbracket core(\mu_m) \rrbracket\}\rho$$
where $\vec{\psi} = BV(\mu_1) \cup \ldots \cup BV(\mu_2)$, and $BV(\mu_1) \ldots BV(\mu_2)$ are distinct from each other and free variables.

$$\begin{aligned}\llbracket \Delta_0, \ldots, \Delta_n \rrbracket_{R_0,\ldots,R_n} = &\{r_0 : \llbracket \Delta_0 \rrbracket, \ldots, r_n : \llbracket \Delta_n \rrbracket\} \mathrel{<\!\!+} \\ &\{z : \llbracket \Delta_0(x) \rrbracket \mid x \in dom(R_0) \wedge z = R_0(x)\} \mathrel{<\!\!+} \ldots \mathrel{<\!\!+} \\ &\{z : \llbracket \Delta_n(x) \rrbracket \mid x \in dom(R_n) \wedge z = R_n(x)\}\end{aligned}$$

Figure 5.14: Translating λ^{gen}_{poly} types to record calculus types.

5.5.1 Type Transformation

The translation in Figure 5.14 converts objects in the λ^{gen}_{poly} type system to objects in the record calculus.

Lemma 5.5.2 (Type translation is well-defined)**.** *Let A be a λ^{gen}_{poly} type. Then $[\![A]\!] \in RLegType$. Similarly, $[\![\Gamma]\!] \in RRec$, for any Γ. Furthermore, for any $B' \in RLegType$, there exists a unique λ^{gen}_{poly} type B such that $[\![B]\!] = B'$. Similarly, for any $\Gamma' \in RType$, there exists a unique Γ such that $[\![\Gamma]\!] = \Gamma'$. Therefore, $[\![\cdot]\!]$ for types is reversible (i.e. is a bijection).*

Proof. Straightforward. □

5.6 Relation Between Staged Programming and Record Calculus

In this section we provide formal properties about the relation between staged computation and the record calculus. We show that the record type system can be used as a sound type system for staged programming, and this type system is as powerful as the λ^{open}_{poly} [KYC06] type system. Although our focus is on typing, we first begin with the theorem which states that evaluating a staged expression in the staged semantics is equivalent to evaluating the expression's translation in the record semantics. In addition to showing the close relation between staged computation and record calculus, this theorem's real value comes into play when proving that the record type system preserves types with respect to staged semantics. The relation between the two operational semantics gives the preservation property *for free*.

Theorem 5.6.1 (Operational Equivalence)**.** *Let e_1 be a stage-n λ^{gen}_{poly} expression such that $FV^n(e_1) = \varnothing$. If $e_1 \longrightarrow_n e_2$, then $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$.*

Proof. By structural induction on e_1, based on the last applied reduction rule. □

We now show that the record type system can be used as a sound type system for the staged language. We follow the standard approach of splitting the soundness into two properties: preservation and progress. Preservation comes for free as a result of Theorem 5.6.1. Progress is explicitly proved.

Later in this section we also prove that the record calculus forms a type system equal to λ^{open}_{poly}. This result suffices to prove soundness of the record type system with respect to staged semantics (because $\lambda^{open}_{poly}$is proven to be sound [KYC06]). However, we prefer to prove soundness via preservation and progress because this would be the approach to take if a variant of a record type system is used for which there is no equal staged type system known. And in fact this is exactly the case for the type system with subtyping (see Section 5.7).

Theorem 5.6.2 (Preservation)**.** *Let e_1 be a stage-n λ^{gen}_{poly} expression such that $FV^n(e_1) = \varnothing$ (i.e. e_1 does not have any stage-0 free variables). If $\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A$ and $e_1 \longrightarrow_n e_2$, then $\Delta \vdash_R [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A$.*

Proof. By Theorem 5.6.1 we have $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$. By the Preservation property of the record type system with respect to the record semantics, we have $\Delta \vdash_R [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A$. □

Lemma 5.6.3. *Let e be a stage-n λ^{gen}_{poly} expression. If $\Delta \vdash_R [\![e]\!]_{R_0,\ldots,R_n} : T$, then $T \in RLegType$.*

Proof. By a straightforward case analysis. □

Theorem 5.6.4 (Progress)**.** *Let e_1 be a stage-n λ^{gen}_{poly} expression. If $\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A$, then either $e_1 \in Val^n$ or there exists e_2 such that $e_1 \longrightarrow_n e_2$.*

Proof. By structural induction on e_1. Lemma 5.6.3 forms a key part in the proof. □

Theorem 5.6.5 (Soundness)**.** *Let e_1 be a stage-0 λ^{gen}_{poly} expression. If $\varnothing \vdash_R [\![e_1]\!]_{\{\}} : A$, then either $e_1 \Uparrow$, or there exists $e_2 \in Val^0$ such that $e_1 \longrightarrow^*_0 e_2$ and $\varnothing \vdash_R [\![e_2]\!]_{\{\}} : A$.*

Proof. Follows from Theorems 5.6.2 and 5.6.4. □

We finally show that the record calculus provides a type system that is the same as λ^{open}_{poly} [KYC06]. This result is important because it shows the power of the type system we obtain via record calculus. Proving only soundness is not sufficient for usefulness — a type system that rejects everything is also sound.

Theorem 5.6.6. *Let e be a stage-n λ^{gen}_{poly} program. Then*

$$\Delta_0, \ldots, \Delta_n \vdash_S e : A \iff [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e]\!]_{R_0,\ldots,R_n} : [\![A]\!]$$

Proof. By structural induction on e. □

5.7 Extending λ^{gen}_{poly} with Subtyping

In Section 5.2 we showed using the library specialization problem why subtyping is needed. In this section we give more details about the need and use of subtyping. We then discuss the existing literature that provides a record type system with subtyping. We finally relate the subtyped record calculus to staged semantics. To distinguish subtyping between record fields from subtyping between types, we use the notation of Rémy [Rém94] in this section. Existing bindings are denoted as `Pre` A, whereas absence is still `Abs`.

5.7.1 Power of Subtyping

A simplification of the library specialization example from Section 5.2 — assuming the existence of tuples — is the function

$$\lambda c.\langle \mathsf{let}\, x = 1 \,\mathsf{in}\, \grave{}(c), \mathsf{let}\, y = 1 \,\mathsf{in}\, \grave{}(c)\rangle$$

Let us call this function G. The best type that λ^{open}_{poly} can give to G is

$$\Box(\{x : \mathtt{Pre}\,\mathtt{int}, y : \mathtt{Pre}\,\mathtt{int}\}\rho \rhd \alpha) \rightarrow \Box(\{x : \mathtt{Pre}\,\mathtt{int}, y : \mathtt{Pre}, \mathtt{int}\}\rho \rhd \alpha * \alpha)$$

Even though this type is sound, it is not satisfactory. Let us now examine through two examples why this type does not suffice and what the ideal type would look like.

- Suppose G is applied on $\langle 0\rangle$. The result of the application would be $\langle \mathsf{let}\, x = 1 \,\mathsf{in}\, 0, \mathsf{let}\, y = 1 \,\mathsf{in}\, 0\rangle$. This fragment does not require any variables from outside; hence it is runnable. Its type ideally would be $\Box(\rho \rhd \mathtt{int} * \mathtt{int})$. However, the λ^{open}_{poly} type for the application is $\Box(\{x : \mathtt{Pre}\,\mathtt{int}, y : \mathtt{Pre}\,\mathtt{int}\}\rho \rhd \mathtt{int} * \mathtt{int})$, which makes unnecessary requirements for x and y, and does not allow us to $\mathsf{run}()$ the value.

- Suppose G is applied on $\langle x + 1\rangle$. The result of the application would be $\langle \mathsf{let}\, x = 1 \,\mathsf{in}\, x + 1, \mathsf{let}\, y = 1 \,\mathsf{in}\, x + 1\rangle$. This fragment requires x to come as an integer value from outside, but imposes no requirements for y. Hence its type ideally would be $\Box(\{x : \mathtt{Pre}\,\mathtt{int}\}\rho \rhd \mathtt{int} * \mathtt{int})$. However, the λ^{open}_{poly} type for the application is again $\Box(\{x : \mathtt{Pre}\,\mathtt{int}, y : \mathtt{Pre}\,\mathtt{int}\}\rho \rhd \mathtt{int} * \mathtt{int})$ which makes an unnecessary requirement for y similar to the case above.

In summary, we do not want the type system to result in code types that put unnecessary requirement on the outer environment. λ^{open}_{poly} does not satisfy this. The technical reason is that c is a parameter of a function and in the Hindley/Milner style let-polymorphism function parameters cannot be given polymorphic types because type checking and inference then becomes undecidable [Wel94, Jim96]. Otherwise we could give a more general type to c and instantiate it accordingly for the two different uses. Because we cannot use a polymorphic type, different uses of the variable have to have the exact same type, resulting in unneeded requirements. This is where subtyping becomes very handy: We can loosen the condition that different uses must have the same type. Suppose c has the type $\Box(\Gamma \rhd \mathtt{int})$. As long as the context of an antiquotation of c provides the contents of Γ, we are fine; there is no harm in providing c with a richer environment than it needs. Suppose also that the outer environment of the fragment $\langle \mathsf{let}\, x = 1 \,\mathsf{in}\, \grave{}(c), \mathsf{let}\, y = 1 \,\mathsf{in}\, \grave{}(c)\rangle$ is Γ'. The environment that goes into the first antiquotation of c is $\Gamma' \Leftarrow \{x : \mathtt{Pre}\,\mathtt{int}\}$ and the second is $\Gamma' \lessdot\!\!+ \{y : \texttt{Pre int}\}$. We want these environments to "satisfy" Γ. That is, we want

$$\Gamma' \lessdot\!\!+ \{x : \texttt{Pre int}\} <: \Gamma \text{ and } \Gamma' \lessdot\!\!+ \{y : \texttt{Pre int}\} <: \Gamma$$

Recall that the environments are nothing but records. Therefore this relation is simply record subtyping [Pie02]: Γ_1 is a subtype of Γ_2 if $\Gamma_1(z)$ is a subtype of $\Gamma_2(z)$ for all $z \in dom(\Gamma_2)$ (thus, $\texttt{Pre}\,A <: \texttt{Abs}$). Because x and y are critical variables, let us write Γ' as $\{x : \theta_1, y : \theta_2\}\rho$ where the field variable θ_i means either absence of the binding or presence of a type. Then, we want Γ to be the least upper bound (lub) of $\{x : \texttt{Pre int}, y : \theta_2\}\rho$ and $\{x : \theta_1, y : \texttt{Pre int}\}\rho$ in the record subtyping lattice. This lub value is $\{x : \theta_1, y : \theta_2\}\rho$ with the condition that $\texttt{Pre int} <: \theta_1$ and $\texttt{Pre int} <: \theta_2$. So, we can give G the type

$$\Box(\{x : \theta_1, y : \theta_2\}\rho \triangleright \alpha) \to \Box(\{x : \theta_1, y : \theta_2\}\rho \triangleright \alpha * \alpha) \text{ where } \texttt{Pre int} <: \theta_1 \text{ and } \texttt{Pre int} <: \theta_2$$

Let us now check if this type can fulfill our needs.

- Suppose G is applied on $\langle 0 \rangle$. The operand has no requirements for x or y. Therefore we can set $\theta_1 = \texttt{Abs}$ and $\theta_2 = \texttt{Abs}$. This satisfies the constraints of the type because $\texttt{Pre int} <: \texttt{Abs}$ and results in $\Box(\{x : \texttt{Abs}, y : \texttt{Abs}\}\rho \triangleright \texttt{int} * \texttt{int})$: a runnable type (simply instantiate ρ with $\varnothing$ and note that $\texttt{Abs}$ stands for the absence of the binding). This is a desired type as mentioned before.

- Suppose G is applied on $\langle x + 1 \rangle$. The operand requires x to come from the outer environment as an integer and has no requirements for y. Therefore we can set $\theta_1 = \texttt{Pre int}$ and $\theta_2 = \texttt{Abs}$. This satisfies the constraints of the type because $\texttt{Pre int} <: \texttt{Pre int}$ and $\texttt{Pre int} <: \texttt{Abs}$, resulting in $\Box(\{x : \texttt{Pre int}, y : \texttt{Abs}\}\rho \triangleright \texttt{int} * \texttt{int})$. Again, a desired type for the application.

To check that the type is sound, suppose we apply G on a fragment that requires y to be a boolean value such as $\langle y \,\&\, \texttt{false} \rangle$. This would set θ_2 to $\texttt{Pre bool}$. The application would result in the fragment $\langle \text{let}\, x = 1 \,\text{in}\, y \,\&\, \texttt{false}, \text{let}\, y = 1 \,\text{in}\, y \,\&\, \texttt{false} \rangle$ which clearly is type-incorrect and should be rejected. With the substitution $\theta_2 = \texttt{Pre bool}$, the constraint $\texttt{Pre int} <: \theta_2$ fails because $\texttt{int}$ is not a subtype of $\texttt{bool}$. Hence the type system would reject the application as expected.

5.7.2 Subtyped Record Calculus

Using a type with subtyping constraints works very well for our purposes. The question is, does there exist any type system that could give us such types? The answer is, fortunately, yes. Odersky, Sulzmann and Wehr define HM(X) [OSW99] which is a Hindler/Milner-style type system parameterized on a constraint system X. Given a constraint system that satisfies the requirements, HM(X) can provide a type system with a principal type inference algorithm for free. Pottier defines SRC, a constraint system that combines subtyping, records, and row variables [Pot00b]. SRC is a sound constraint system in the style of [OSW99], and yields the type system HM(SRC).

Taking advantage of the close relation between staged computation and record calculus, we can use HM(SRC) to type-check staged expressions after translating them to the record calculus. The translation is the same.

SRC is a very powerful constraint system; it is more powerful than we need. It provides *conditional constraints* that handle the tricky record concatenation problem. We do not need record concatenation. Handling record extension suffices in our context. So we ignore conditional constraints. SRC is parameterized on a *ground signature*. Pottier defines a sample ground signature in [Pot00b] and uses the resulting system to obtain a type system that can type-check accurate pattern matching, record concatenation and first-class messages using a single framework. We use the same ground signature Pottier defines. Below are the modifications we need to make to the definition of types in Figure 5.11. This definition is then fed into SRC to obtain a type system with record subtyping and row variables. It is straightforward to check that these definitions preserve the properties of the ground signature.

$$\begin{array}{rcl} T & \in & RType ::= \ldots \mid \top \mid \bot \\ F & \in & RField ::= \ldots \mid \texttt{bot} \end{array}$$

The modifications are straightforward. Pottier requires types to form a lattice where the smallest and greatest elements are nullary. Therefore we add $\top$ and a $\bot$ to the types, and bot to the definition of fields. The ordering between types is standard and the same as in [Pot00b]; the left-hand-side of a function type is contravariant, its right-hand-side and record types are covariant. Fields are ordered as $\texttt{bot} < \texttt{Pre}\,A < \texttt{Abs}$. Note that this ordering is simpler than Pottier's, where Abs and $\texttt{Pre}\,A$ are incomparable and have a common upper value, $\texttt{Either}\,A$. Pottier uses that ordering again to handle the record concatenation problem. Because we do not need concatenation, a simple chain ordering suffices.

HM(X) assumes core ML as the syntax of its language. New syntax can be treated as function applications where the functions are kept in a pervasive environment. In HM(SRC), record operations have the following types.

$$\begin{array}{rcl} \{\} & : & \{\} \\ \_\cdot a & : & \forall \alpha, \rho. \{a : \texttt{Pre}\,\alpha\}\rho \to \alpha \\ \_\,\textsf{with}\,\{a = \_\} & : & \forall \theta, \alpha, \rho. \{a : \theta\}\rho \to \alpha \to \{a : \texttt{Pre}\,\alpha\}\rho \end{array}$$

In the record language we distinguished record variables from regular variables in order to keep the type system sound with respect to staged semantics. We need to do the same in HM(SRC). We do not elaborate this issue since it is straightforward.

A question arises for the requirement of a lattice in the definition of types. Suppose f is a function with two applications: $f(1)$ and $f(\texttt{true})$. Let the input type of f be α. Type inference collects for α the constraints $\texttt{int} <: \alpha$ and $\texttt{bool} <: \alpha$, which give $\alpha = \top$ assuming a standard flat lattice, and the type is accepted by the type system. However, this could be considered as an ill-typed situation by many type systems. Even though Pottier requires types to form a lattice, this is not a requirement for the soundness of the constraint system, but for the correctness of constraint simplification algorithms which improve readability of constraints that are attached to types. There are simplification algorithms that do not impose a lattice structure but are less efficient than Pottier's [Fre97, Reh98], as well as other record subtype systems that do not assume lattices [EST95].

Nanevski [Nan02] defines a staged language where the type of a fragment contains the free variables of the fragment, called the "support set". A subtyping rule is defined, making it possible to use a code fragment in a context that provides more variables than required. In other words, subtyping loosens the support set of a code value. While this idea is very useful, it does not provide a general solution to the subtyping problem we cover here, because a support set contains only the names of the variables; no type information is stored. Hence, we can only reason about existence or absence of a variable in the support set. On the other hand, Pottier's subtyping constraints give the ability to keep the types related to variables and also the subtyping relations between these types. Therefore, Pottier's system subsumes Nanevski's definition of subtyping in our context.

5.7.3 Implementation

Pottier provides an implementation of his subtyped constraint system, called Wallace [Pot00a]. He also provides an implementation of a type system for a toy programming language that supports record operations. To experiment with the ideas we discussed about subtyping, we have implemented the translation and the type inference algorithm of λ^{open}_{poly}. We then type-checked the resulting translations in Wallace. The types output by our implementation and Wallce are in conformance with our expectations. A screenshot of our test for the example discussed in Section 5.7.1 is given in Figure 5.15.

5.7.4 Staged Semantics and Subtyped Record Calculus

Subtyping is about being able to replace a value of some type with another value of subtype without sacrificing safety. We mentioned above that it is okay to supply a "richer" environment to a code fragment than the environment it expects. Suppose the code fragment's type is $\Box(\Gamma \triangleright A)$ and it is provided with the environment Γ'. Then, what we really want is that the $\Box(\Gamma \triangleright A)$ behave like $\Box(\Gamma' \triangleright A)$. That is

$$\Box(\Gamma \triangleright A) <: \Box(\Gamma' \triangleright A)$$

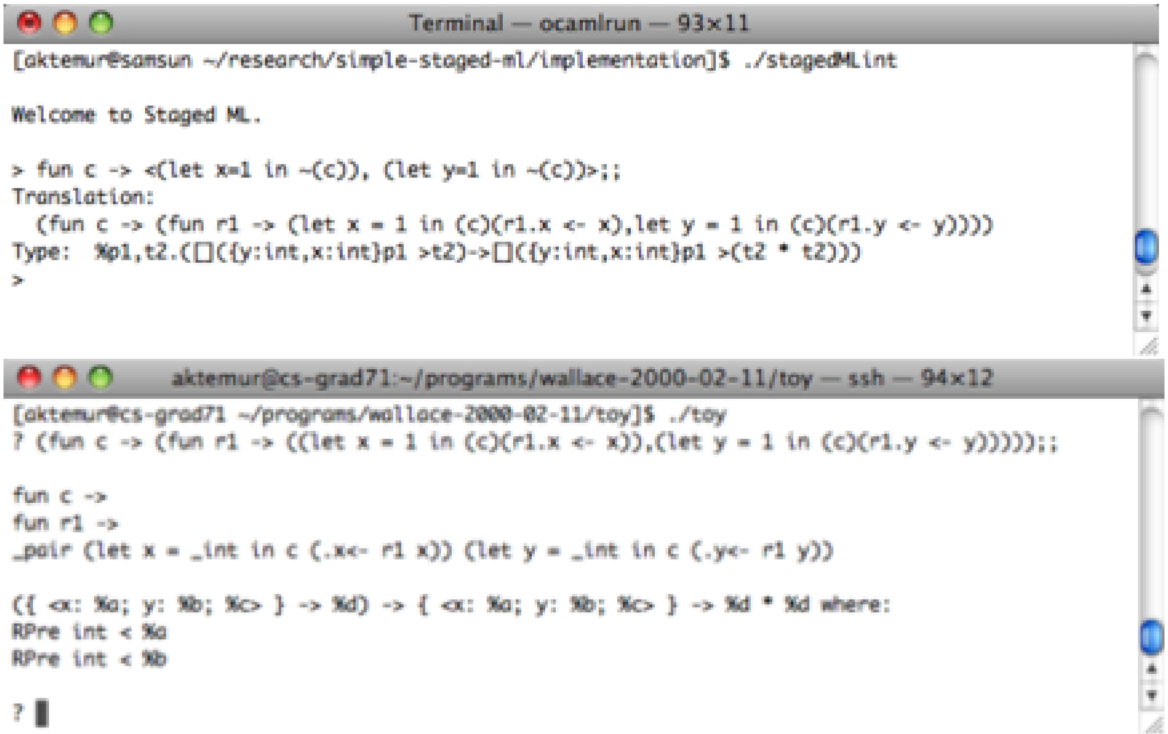

Figure 5.15: The first picture above is a screenshot of our implementation of the translation and the λ^{open}_{poly} type system. The example being tested is the one discussed in Section 5.7.1. Here, an ASCII representation of the type is given, where "%" stands for the $\forall$ symbol, and ">" stands for the $\triangleright$ symbol. The second picture is Pottier's implementation of the record type system with subtyping constraints, called Wallace. "`e.x<-e'`" is Pottier's syntax for "`e with x=e'`". In Wallace's types, "%" denotes the quantified variables, and "<" is the subtyping relation. The type reported by Wallace is exactly the one we expect, as discussed in Section 5.7.1.

Because Γ' is richer than Γ, we have $\Gamma' <: \Gamma$. This suggests that the subtyping relation for the environment component of a $\Box$-type is *contravariant*. With a similar reasoning, it is easy to find that the relation for the type part is *covariant*. So we have

$$\frac{\Gamma' <: \Gamma \quad A <: A'}{\Box(\Gamma \triangleright A) <: \Box(\Gamma' \triangleright A')}$$

Recall that $\Box$-types are translated to function types. That is, $[\![\Box(\Gamma \triangleright A)]\!] = [\![\Gamma]\!] \to [\![A]\!]$, where the left-hand-side type is contravariant and the right-hand-side is covariant [Pie02]. Therefore, subtyping relations are preserved by the translation.

We can again use the record calculus type system to type-check staged expressions. The following state related properties. We first state that the record type system with subtyping is sound with respect to staged semantics. This is done via Preservation and Progress again.

Theorem 5.7.1 (Preservation). *Let e_1 be a stage-n λ^{gen}_{poly} expression such that $FV^n(e_1) = \varnothing$ (i.e. e_1 does not have any stage-0 free variables). If $e_1 \longrightarrow_n e_2$ and $[\![e_1]\!]_{\{\},R_1,\ldots,R_n}$ is typable in HM(SRC), then $[\![e_2]\!]_{\{\},R_1,\ldots,R_n}$ is also typable in HM(SRC) to the same type under the same assumptions.*

Proof. By Theorem 5.6.1 we have $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$. Because HM(SRC) is a sound type system, it has the Preservation property. Therefore, $[\![e_2]\!]_{\{\},R_1,\ldots,R_n}$ can be given the same type of $[\![e_1]\!]_{\{\},R_1,\ldots,R_n}$. □

Theorem 5.7.2 (Progress). *Let e_1 be a stage-n λ^{gen}_{poly} expression. If $[\![e_1]\!]_{\{\},R_1,\ldots,R_n}$ is typable in HM(SRC), then either $e_1 \in Val^n$ or there exists e_2 such that $e_1 \longrightarrow_n e_2$.*

Proof. Similar to Theorem 5.6.4, by reverse reasoning about the structure of types that a result of the translation can get. □

Theorem 5.7.3 (Soundness). *HM(SRC) is a sound type system with respect to staged semantics.*

Proof. By Theorems 5.7.1 and 5.7.2. □

The theorem below says that anything typable in the λ^{open}_{poly} type system is also typable in the record calculus with subtyping. As illustrated by the library specialization example, subtyped record calculus can type more expressions.

Theorem 5.7.4. *Let e be a stage-n λ^{gen}_{poly} program. If $\Delta_0, \ldots, \Delta_n \vdash_S e : A$ then in HM(SRC), $[\![e]\!]_{R_0,\ldots,R_n}$ is typable to $[\![A]\!]$ with no constraints under the environment $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n}$.*

Proof. By Theorem 5.6.6, we have $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e]\!]_{R_0,\ldots,R_n} : [\![A]\!]$. Because record calculus with subtyping subsumes record calculus without subtyping, the same judgment holds in HM(SRC), too. □

We do not give a definition for a standalone staged type system that has subtyping. We leave it as future work.

5.8 Extending λ^{gen}_{poly} with Pluggable Declarations

In this section we discuss how we can extend the staged language with pluggable declarations. Additions to the syntax and static and dynamic semantics are listed in Figure 5.16. We refer to this language as λ^{decl}_{poly}. We use a type of the form $\Diamond(\Gamma \triangleright \Gamma')$ for quoted declarations. The choice of the symbol $\Diamond$ is arbitrary; it is not related to another usage in other areas of mathematic, in particular as the *possibility* operator in modal logic. In fact, the modal property of $\Diamond$ is the same as $\Box$, which is briefly discussed by Kim, Yi and Calcagno [KYC06, §3.3]: If e has type $\Diamond(\Gamma \triangleright \Gamma')$, then $`(e)$ (as part of a let-binding $\mathsf{let}\ `(e)\ \mathsf{in}\ e'$) produces the environment Γ' if used in an environment satisfying Γ.

Syntax

$$Exp ::= \ldots \mid \langle\rangle \mid \langle x = e\rangle \mid \mathsf{let}\ \grave{}(e)\ \mathsf{in}\ e$$

Values

$$\begin{array}{lll} Val^0 & ::= \ldots \mid \langle\rangle \mid \langle x = v^1\rangle & \\ Val^{n+1} & ::= \ldots \mid \langle\rangle \mid \langle x = v^{n+2}\rangle & \\ & \mid \mathsf{let}\ \grave{}(v^n)\ \mathsf{in}\ v^{n+1} & (\text{if } n > 0) \end{array}$$

Operational Rules

$$\text{ESDEC} \quad \frac{e \longrightarrow_{n+1} e'}{\langle x = e\rangle \longrightarrow_n \langle x = e'\rangle}$$

$$\text{ESLET2} \quad \frac{e_1 \longrightarrow_n e_1'}{\mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2 \longrightarrow_{n+1} \mathsf{let}\ \grave{}(e_1')\ \mathsf{in}\ e_2}$$

$$\frac{e_1 \in Val^n \quad e_2 \longrightarrow_{n+1} e_2'}{\mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2 \longrightarrow_{n+1} \mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2'}$$

$$\frac{e_1 \in Val^1 \quad e_2 \in Val^1}{\mathsf{let}\ \grave{}(\langle x = e_1\rangle)\ \mathsf{in}\ e_2 \longrightarrow_1 \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2}$$

$$\frac{e_2 \in Val^1}{\mathsf{let}\ \grave{}(\langle\rangle)\ \mathsf{in}\ e_2 \longrightarrow_1 e_2}$$

Types

$$SType ::= \ldots \mid \Diamond(\Gamma \triangleright \Gamma')$$

Typing Rules

$$\text{TSEDEC} \quad \Delta_0, \ldots, \Delta_n \vdash_P \langle\rangle : \Diamond(\Gamma \triangleright \Gamma)$$

$$\text{TSDEC} \quad \frac{\Delta_0, \ldots, \Delta_n, \Gamma \vdash_P e : A}{\Delta_0, \ldots, \Delta_n \vdash_P \langle x = e\rangle : \Diamond(\Gamma \triangleright \Gamma \Leftarrow\!\!\!+ \{x : A\})}$$

$$\text{TSLET2} \quad \frac{\begin{array}{c}\Delta_0, \ldots, \Delta_n \vdash_P e_1 : \Diamond(\Gamma \triangleright \Gamma') \quad \Gamma \prec \Delta_{n+1} \\ \Delta_0, \ldots, \Delta_n, \Gamma' \vdash_P e_2 : A\end{array}}{\Delta_0, \ldots, \Delta_n, \Delta_{n+1} \vdash_P \mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2 : A}$$

Other typing rules are copied from λ^{gen}_{poly}.

Other Definitions

$$\begin{array}{rcl} FV^n(\langle x = e\rangle) & = & FV^{n+1}(e) \\ FV^{n+1}(\mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2) & = & FV^n(e_1) \cup FV^{n+1}(e_2) \\ \langle y = e\rangle[x \backslash e']^n & = & \langle y = e[x \backslash e']^{n+1}\rangle \\ (\mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2)[x \backslash e']^{n+1} & = & \mathsf{let}\ \grave{}(e_1[x \backslash e']^n)\ \mathsf{in}\ e_2[x \backslash e']^{n+1} \\ \langle x = e\rangle[y^m \overset{n}{\mapsto} z] & = & \langle x = e[y^m \overset{n+1}{\mapsto} z]\rangle \\ (\mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2)[y^m \overset{n+1}{\mapsto} z] & = & \mathsf{let}\ \grave{}(e_1[y^m \overset{n}{\mapsto} z])\ \mathsf{in}\ e_2[y^m \overset{n+1}{\mapsto} z] \end{array}$$

Figure 5.16: Extending λ^{gen}_{poly} with pluggable declarations. We refer to the resulting language as λ^{decl}_{poly}.

In this section we show that the extension with pluggable declarations retains soundness of the staged type system. We also show that pluggable declarations are syntactic sugaring. We finally discuss how the translation into the record calculus is affected.

5.8.1 Soundness of the λ^{decl}_{poly} Type System

We now show that the new staged type system that handles pluggable declarations (or bindings, in the ML terminology) is sound. The original staged type system, λ^{gen}_{poly}, was already proven sound in [KYC06]. We show that the Preservation and Progress are still valid.

Theorem 5.8.1 (Preservation)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression. If $\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P e_1 : A$ and $e_1 \longrightarrow_n e_2$, then $\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P e_2 : A$.*

Proof. By structural induction on e_1. □

Theorem 5.8.2 (Progress)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression. If $\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P e_1 : A$, then either $e_1 \in Val^n$ or there exists e_2 such that $e_1 \longrightarrow_n e_2$.*

Proof. By structural induction on e_1. □

5.8.2 Pluggable Declarations are Syntactic Sugar

We now show that pluggable declarations are syntactic sugaring; what we can express using them can already be expressed using the existing quotation/antiquotation mechanism.

First, define the following desugaring function, $\delta(\cdot)$, from λ^{decl}_{poly} expressions to λ^{gen}_{poly} expressions. (Cases not shown simply recurse into subexpressions.)

$$
\begin{array}{rcl}
\delta(\langle\rangle) & = & \lambda y.\langle\,\grave{}(y)\rangle \\
\delta(\langle x = e\rangle) & = & (\lambda v.\lambda y.\langle \mathsf{let}\, x = \,\grave{}(v)\, \mathsf{in}\, \grave{}(y)\rangle)\ \langle\delta(e)\rangle \\
\delta(\mathsf{let}\, \grave{}(e_1)\, \mathsf{in}\, e_2) & = & \grave{}(\delta(e_1)\ \langle\delta(e_2)\rangle)
\end{array}
$$

Note that the desugaring function for the quoted declaration $\langle x = e\rangle$ produces a function application where $\langle\delta(e)\rangle$ is the operand. Not placing it inside the λ-abstraction allows for any antiquotations to be evaluated, which would not be possible under the abstraction at stage-0.

Also define the desugaring function below from λ^{decl}_{poly} types to λ^{gen}_{poly} types. Cases not shown simply recurse into sub-components.

$$\delta(\Diamond(\Gamma_1 \triangleright \Gamma_2)) = \Box(\delta(\Gamma_2) \triangleright A) \to \Box(\delta(\Gamma_1) \triangleright A) \text{ for any type } A.$$

The following theorem states that desugaring preserves operational semantics; evaluation of an expression with pluggable declarations is equivalent to evaluating its desugared version.

Theorem 5.8.3. *Let e_1 be a λ^{decl}_{poly} expression such that $e_1 \longrightarrow_n e_2$. Then $\delta(e_1) \longrightarrow^*_n \delta(e_2)$.*

In addition to operational semantics, we also show that desugaring preserves typing. The theorem below states that anything typable in λ^{decl}_{poly} is also typable in λ^{gen}_{poly} after desugaring.

Theorem 5.8.4. $\Delta_0, \ldots, \Delta_n \vdash_P e : A \implies \delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \delta(e) : \delta(A)$

Proof. By structural induction on e. □

Note that although the desugaring function $\delta(\cdot)$ is surjective, it is not injective. This prevents the desugaring from being reversible. This is the reason why the relation shown in Theorem 5.8.4 is one-directional ($\Longrightarrow$) instead of an if-and-only-if ($\Longleftrightarrow$) relation.

The properties shown in Theorems 5.8.3 and 5.8.4 mean that anything expressible using pluggable declarations is also expressible using core program generation facilities (i.e. quotable expressions). In other words, pluggable declarations do not bring extra expressive power. Despite this fact, they are useful because they eliminate the need to use higher-order functions which may make a program hard to understand and manipulate for programmers.

5.8.3 Translation into Record Calculus

The fact that a declaration is syntactically not an expression but that it becomes an expression when quoted brings a problem in typing. We first explain the problem, then elaborate a solution.

Analogous to a quoted expression, the immediate idea is to represent a quoted declaration as a function that takes in an environment as its input. As the output, the function produces another environment. More concretely, the quoted declaration $\langle x = e\rangle$ at stage n would be translated to the function $\lambda r.r\,\mathsf{with}\,\{x = e'\}$ where e' is the translation of e. This function would have to be given a type of the form $\Gamma_1 \to \Gamma_2$. However, this type cannot be constructed from the definition in Figure 5.11 given for the record type system. Adding this type to the grammar of types introduces the problems mentioned in Section 5.4.3. To overcome this problem, we can translate quoted declarations to higher order functions similar to desugaring. For instance, the declaration $\langle x = 1\rangle$ would first be converted to $\lambda y.\langle \mathsf{let}\, x = 1\, \mathsf{in}\ \grave{}(y)\rangle$ and then translated to record calculus. However, this translation brings another problem. Misuses of declarations as functions cannot be detected; e.g. $\langle x = 1\rangle\langle 0\rangle$ would pass the type checker. To overcome this problem, we translate declarations to functions that take a unique value as input; let-bindings with holes then apply the antiquoted declaration to this unique value. This way we can distinguish declarations from quoted expressions or lambda abstractions because the translation of a declaration cannot be applied to anything but the unique value. For this purpose, we add to the record language a special variable and constant κ that has type κ. This constant and its type do not exist in the staged language. We then extend the definition of record types as follows:

$$T \quad \in \quad RType ::= \ldots \mid \kappa$$

The translation function is then extended with the following definitions.

$[\![\langle\rangle]\!]_{R_0,\ldots,R_n} = \lambda\kappa.\lambda y.\lambda r.\, y(r)$

$[\![\langle x = e\rangle]\!]_{R_0,\ldots,R_n} = \lambda\kappa.\lambda y.\lambda r.\, \mathsf{let}\, z = [\![e]\!]_{R_0,\ldots,R_n,r}\, \mathsf{in}\, y(r\, \mathsf{with}\, \{x = z\})$ where r, y, z are fresh.

$[\![\mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2]\!]_{R_0,\ldots,R_{n+1}} = ([\![e_1]\!]_{R_0,\ldots,R_n})\kappa\, (\lambda r.[\![e_2]\!]_{R_0,\ldots,R_n,r})\, R_{n+1}$ where r is fresh.

We now show that the record calculus provides a sound type system with respect to λ^{decl}_{poly} operational semantics. Following the same approach we did for λ^{gen}_{poly}, we first state the theorem about the relation between two operational semantics, which provides the preservation theorem for free. This is followed by the progress theorem.

Theorem 5.8.5 (Operational Equivalence)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression such that $FV^n(e_1) = \varnothing$. If $e_1 \longrightarrow_n e_2$, then $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$.*

Proof. By structural induction on e_1, based on the last applied reduction rule. □

Theorem 5.8.6 (Preservation)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression. If $\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A$ and $e_1 \longrightarrow_n e_2$, then $\Delta \vdash_R [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A$.*

Proof. By the same proof method we used in Theorem 5.6.2: By Theorem 5.8.5 we have $[\![e_1]\!]_{\{\},R_1,\ldots,R_n} \longrightarrow^*_\beta [\![e_2]\!]_{\{\},R_1,\ldots,R_n}$. By the Preservation property of the record type system with respect to the record semantics, we have $\Delta \vdash_R [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A$. □

Theorem 5.8.7 (Progress)**.** *Let e_1 be a stage-n λ^{decl}_{poly} expression. If $\Delta \vdash_R [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A$, then either $e_1 \in Val^n$ or there exists e_2 such that $e_1 \longrightarrow_n e_2$.*

Proof. By structural induction on e. □

Theorem 5.8.8 (Soundness)**.** *Let e_1 be a stage-0 λ^{decl}_{poly} expression. If $\varnothing \vdash_R [\![e_1]\!]_{\{\}} : A$, then either $e_1 \Uparrow$, or there exists $e_2 \in Val^0$ such that $e_1 \longrightarrow^*_0 e_2$ and $\varnothing \vdash_R [\![e_2]\!]_{\{\}} : A$.*

Proof. Follows from Theorems 5.8.6 and 5.8.7. □

We finally show that using the record type system to type-check λ^{decl}_{poly} expression yields a type system that is as powerful as the λ^{decl}_{poly} type system. We first extend the definition of type translation:

$$[\![\Diamond(\Gamma_1 \triangleright \Gamma_2)]\!] = \kappa \rightarrow ([\![\Gamma_2]\!] \rightarrow B) \rightarrow ([\![\Gamma_1]\!] \rightarrow B) \text{ for any } B$$

Note that the first half of Lemma 5.5.2 still holds. That is, for any λ_{poly}^{decl} type A, $[\![A]\!] \in RLegType$. However, the backwards direction, which says that for any $A' \in RLegType$ there exists a λ_{poly}^{gen} type A such that $[\![A]\!] = A'$, is no longer valid due to the extension of the type system with κ. Because of this fact, the relation between λ_{poly}^{decl} type system and record type system is no longer bi-directional (i.e. not an iff relation). The relation we have now says that anything typable in λ_{poly}^{decl} is also typable in the record calculus. This property essentially means that record calculus provides a type system as powerful as λ_{poly}^{decl}, and is much more important than the other direction.

Theorem 5.8.9. *Let e be a stage-n λ_{poly}^{decl} program. Then*

$$\Delta_0, \ldots, \Delta_n \vdash_P e : A \implies [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e]\!]_{R_0,\ldots,R_n} : [\![A]\!]$$

Proof. By structural induction on e. □

In conclusion, the record calculus provides a sound type system that is as powerful as λ_{poly}^{decl}.

5.9 Extending λ_{poly}^{gen} with References

The order of evaluation in the (call-by-value) semantics of λ_{poly}^{gen}, which is given in Figure 5.7, is not preserved by the translation given in Figure 5.13, if the result of the translation is evaluated using standard call-by-value semantics of the record calculus: In the staged semantics holes in a quoted fragment are evaluated first. However, the translation converts a quoted expression to a lambda abstraction which would immediately evaluate to a closure, giving the behavior that the holes would be evaluated only when the quoted expression is "run". Because we did not have any side-effects in the language (and because non-termination is ignored by the type system due to undecidability), this difference in the order of evaluation did not matter (for the very same reason we postponed formal definition of the call-by-value record semantics). The translation, however, would be problematic in the presence of side effects. Consider the stage-0 expression $\langle x + \grave{}(\mathsf{ref}\ 0; \langle 1 \rangle)\rangle$. Its transformation would be $\lambda r_1.(r_1 {\cdot} x + (\mathsf{ref}\ 0; \lambda r.1) r_1)$. Executing the staged expression evaluates the hole, resulting in a new memory allocation. On the other hand, its translation is an abstraction, and immediately evaluates to a closure without expanding the memory.

In this section we first add references to the record calculus and the staged language. We then modify the translation to preserve the order of execution and show several formal properties. We conclude with a discussion of how to handle pluggable declarations in the presence of references.

$$Val^0 ::= \ldots \mid \ell$$
$$Val^{n+1} ::= \ldots \mid \ell \mid \mathsf{ref}\ v^{n+1} \mid !v^{n+1} \mid v^{n+1}{:=}v^{n+1}$$
$$\mathcal{S} \in \mathit{Store} = \mathit{Location} \rightharpoonup Val^0$$

ESABS $\dfrac{\mathcal{S}, e \longrightarrow_{n+1} \mathcal{S}', e'}{\mathcal{S}, \lambda x.e \longrightarrow_{n+1} \mathcal{S}', \lambda x.e'}$ ESSYM $\dfrac{z \text{ is fresh}}{\mathcal{S}, \lambda^* x.e \longrightarrow_n \mathcal{S}, \lambda z.[x^n \overset{n}{\mapsto} z]e}$

ESFIX $\dfrac{\mathcal{S}, e \longrightarrow_{n+1} \mathcal{S}', e'}{\mathcal{S}, \mathsf{fix}\, f(x).\, e \longrightarrow_{n+1} \mathcal{S}', \mathsf{fix}\, f(x).\, e'}$

ESAPP $\dfrac{\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_1'}{\mathcal{S}, e_1 e_2 \longrightarrow_n \mathcal{S}', e_1' e_2}$ $\dfrac{e_1 \in Val^n \quad \mathcal{S}, e_2 \longrightarrow_n \mathcal{S}', e_2'}{\mathcal{S}, e_1 e_2 \longrightarrow_n \mathcal{S}', e_1 e_2'}$ $\dfrac{e_2 \in Val^0}{\mathcal{S}, (\lambda x.e)e_2 \longrightarrow_0 \mathcal{S}, e[x\backslash e_2]^0}$

$$\dfrac{e_2 \in Val^0}{\mathcal{S}, (\mathsf{fix}\, f(x).\, e)e_2 \longrightarrow_0 \mathcal{S}, e[f\backslash \mathsf{fix}\, f(x).\, e]^0[x\backslash e_2]^0}$$

ESLET $\dfrac{\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_1'}{\mathcal{S}, \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \longrightarrow_n \mathcal{S}', \mathsf{let}\, x = e_1' \,\mathsf{in}\, e_2}$ $\dfrac{e_1 \in Val^0}{\mathcal{S}, \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \longrightarrow_0 \mathcal{S}, e_2[x\backslash e_1]^0}$

$$\dfrac{e_1 \in Val^{n+1} \quad \mathcal{S}, e_2 \longrightarrow_{n+1} \mathcal{S}', e_2'}{\mathcal{S}, \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \longrightarrow_{n+1} \mathcal{S}', \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2'}$$

ESBOX $\dfrac{\mathcal{S}, e \longrightarrow_{n+1} \mathcal{S}', e'}{\mathcal{S}, \langle e \rangle \longrightarrow_n \mathcal{S}', \langle e' \rangle}$

ESUBOX $\dfrac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, \text{`}(e) \longrightarrow_{n+1} \mathcal{S}', \text{`}(e')}$ $\dfrac{e \in Val^1}{\mathcal{S}, \text{`}(\langle e \rangle) \longrightarrow_1 \mathcal{S}, e}$

ESRUN $\dfrac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, \mathsf{run}(e) \longrightarrow_n \mathcal{S}', \mathsf{run}(e')}$ $\dfrac{e \in Val^1}{\mathcal{S}, \mathsf{run}(\langle e \rangle) \longrightarrow_0 \mathcal{S}, e}$

ESLIFT $\dfrac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, \mathsf{lift}(e) \longrightarrow_n \mathcal{S}', \mathsf{lift}(e')}$ $\dfrac{e \in Val^0}{\mathcal{S}, \mathsf{lift}(e) \longrightarrow_0 \mathcal{S}, \langle e \rangle}$

ESREF $\dfrac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, \mathsf{ref}\ e \longrightarrow_n \mathcal{S}', \mathsf{ref}\ e'}$ $\dfrac{e \in Val^0 \quad \ell \notin dom(\mathcal{S})}{\mathcal{S}, \mathsf{ref}\ e \longrightarrow_0 \mathcal{S} \Leftarrow \{\ell : e\}, \ell}$

ESDEREF $\dfrac{\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'}{\mathcal{S}, !e \longrightarrow_n \mathcal{S}', !e'}$ $\dfrac{\mathcal{S}(\ell) = v}{\mathcal{S}, !\ell \longrightarrow_0 \mathcal{S}, v}$

ESASGN $\dfrac{\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_1'}{\mathcal{S}, e_1{:=}e_2 \longrightarrow_n \mathcal{S}', e_1'{:=}e_2}$ $\dfrac{e_1 \in Val^n \quad \mathcal{S}, e_2 \longrightarrow_n \mathcal{S}', e_2'}{\mathcal{S}, e_1{:=}e_2 \longrightarrow_n \mathcal{S}', e_1{:=}e_2'}$ $\dfrac{e_2 \in Val^0}{\mathcal{S}, \ell{:=}e_2 \longrightarrow_0 \mathcal{S} \Leftarrow \{\ell : e_2\}, e_2}$

Figure 5.17: The operational semantics of λ^{open}_{poly} with references.

TSREF $\dfrac{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S e : A}{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S \mathsf{ref}\ e : A\ \mathsf{ref}}$ TSDEREF $\dfrac{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S e : A\ \mathsf{ref}}{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S !e : A}$

TSASGN $\dfrac{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S e_1 : A\ \mathsf{ref} \qquad \Sigma; \Delta_0, \ldots, \Delta_n \vdash_S e_2 : A}{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S e_1{:=}e_2 : A}$ TSLOC $\dfrac{\Sigma(\ell) = A}{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S \ell : A\ \mathsf{ref}}$

TSLETIMP $\dfrac{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S e_1 : A \quad \text{expansive}^n(e_1) \qquad \Sigma; \Delta_0, \ldots, \Delta_n \Leftarrow \{x : A\} \vdash_S e_2 : B}{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : B}$

TSLETAPP $\dfrac{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S e_1 : A \quad \neg\text{expansive}^n(e_1) \qquad \Sigma; \Delta_0, \ldots, \Delta_n \Leftarrow \{x : \mathrm{GEN}_A(\Sigma, \Delta_0, \ldots, \Delta_n)\} \vdash_S e_2 : B}{\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : B}$

Figure 5.18: The λ^{open}_{poly} typing rules to handle references. Other rules are the same as before except propagating the store typing.

$$v \in RVal ::= \ldots \mid \ell$$
$$\mathcal{S} \in RStore = Location \rightharpoonup RVal$$

ERAPP $\dfrac{\mathcal{S}, e_1 \longrightarrow_R \mathcal{S}', e_1'}{\mathcal{S}, e_1 e_2 \longrightarrow_R \mathcal{S}', e_1' e_2}$ $\quad$ $\dfrac{e_1 \in RVal \quad \mathcal{S}, e_2 \longrightarrow_R \mathcal{S}', e_2'}{\mathcal{S}, e_1 e_2 \longrightarrow_R \mathcal{S}', e_1 e_2'}$ $\quad$ $\dfrac{e_2 \in RVal}{\mathcal{S}, (\lambda w.e_1)e_2 \longrightarrow_R \mathcal{S}, e_1[w \backslash e_2]}$

$$\dfrac{e_2 \in RVal}{\mathcal{S}, (\mathsf{fix}\, f(x).\, e_1)e_2 \longrightarrow_R \mathcal{S}, e_1[f \backslash \mathsf{fix}\, f(x).\, e_1][x \backslash e_2]}$$

ERLET $\dfrac{\mathcal{S}, e_1 \longrightarrow_R \mathcal{S}', e_1'}{\mathcal{S}, \mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2 \longrightarrow_R \mathcal{S}', \mathsf{let}\, w = e_1' \,\mathsf{in}\, e_2}$ $\quad$ $\dfrac{e_1 \in RVal}{\mathcal{S}, \mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2 \longrightarrow_R \mathcal{S}, e_2[w \backslash e_1]}$

ERUPD $\dfrac{\mathcal{S}, e_1 \longrightarrow_R \mathcal{S}', e_1'}{\mathcal{S}, e_1 \,\mathsf{with}\, \{a = e_2\} \longrightarrow_R \mathcal{S}', e_1' \,\mathsf{with}\, \{a = e_2\}}$ $\quad$ $\dfrac{e_1 \in RVal \quad \mathcal{S}, e_2 \longrightarrow_R \mathcal{S}', e_2'}{\mathcal{S}, e_1 \,\mathsf{with}\, \{a = e_2\} \longrightarrow_R \mathcal{S}', e_1 \,\mathsf{with}\, \{a = e_2'\}}$

$$\dfrac{e_2 \in RVal}{\mathcal{S}, \{a_j : v_j\}_1^m \,\mathsf{with}\, \{a = e_2\} \longrightarrow_R \mathcal{S}, \{a_j : v_j\}_1^m \Leftarrow\!\!+ \{a : e_2\}}$$

ERACC $\dfrac{\mathcal{S}, e \longrightarrow_R \mathcal{S}', e'}{\mathcal{S}, e \cdot a \longrightarrow_R \mathcal{S}', e' \cdot a}$ $\quad$ $\mathcal{S}, \{a_j : v_j\}_1^m \cdot a_i \longrightarrow_R \mathcal{S}, v_i$

ERREF $\dfrac{\mathcal{S}, e \longrightarrow_R \mathcal{S}', e'}{\mathcal{S}, \mathsf{ref}\; e \longrightarrow_R \mathcal{S}', \mathsf{ref}\; e'}$ $\quad$ $\dfrac{e \in RVal \quad \ell \notin dom(\mathcal{S})}{\mathcal{S}, \mathsf{ref}\; e \longrightarrow_R \mathcal{S} \Leftarrow\!\!+ \{\ell : e\}, \ell}$

ERDEREF $\dfrac{\mathcal{S}, e \longrightarrow_R \mathcal{S}', e'}{\mathcal{S}, !e \longrightarrow_R \mathcal{S}', !e'}$ $\quad$ $\dfrac{\mathcal{S}(\ell) = v}{\mathcal{S}, !\ell \longrightarrow_R \mathcal{S}, v}$

ERASGN $\dfrac{\mathcal{S}, e_1 \longrightarrow_R \mathcal{S}', e_1'}{\mathcal{S}, e_1{:=}e_2 \longrightarrow_R \mathcal{S}', e_1'{:=}e_2}$ $\quad$ $\dfrac{e_1 \in RVal \quad \mathcal{S}, e_2 \longrightarrow_R \mathcal{S}', e_2'}{\mathcal{S}, e_1{:=}e_2 \longrightarrow_R \mathcal{S}', e_1{:=}e_2'}$

$$\dfrac{e_2 \in RVal}{\mathcal{S}, \ell{:=}e_2 \longrightarrow_R \mathcal{S} \Leftarrow\!\!+ \{\ell : e_2\}, e_2}$$

Figure 5.19: The operational semantics of record calculus with references.

TRREF $\dfrac{\Sigma; \Delta \vdash_R e : A}{\Sigma; \Delta \vdash_R \mathsf{ref}\; e : A\; \mathsf{ref}}$

TRDEREF $\dfrac{\Sigma; \Delta \vdash_R e : A\; \mathsf{ref}}{\Sigma; \Delta \vdash_R !e : A}$

TRASGN $\dfrac{\Sigma; \Delta \vdash_R e_1 : A\; \mathsf{ref} \qquad \Sigma; \Delta \vdash_R e_2 : A}{\Sigma; \Delta \vdash_R e_1{:=}e_2 : A}$

TRLOC $\dfrac{\Sigma(\ell) = A}{\Sigma; \Delta \vdash_R \ell : A\; \mathsf{ref}}$

TRLETIMP $\dfrac{\Sigma; \Delta \vdash_R e_1 : A \quad \text{expansive}(e_1) \qquad \Sigma; \Delta \Leftarrow\!\!+ \{x : A\} \vdash_R e_2 : B}{\Sigma; \Delta \vdash_R \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : B}$

TRLETAPP $\dfrac{\Sigma; \Delta \vdash_R e_1 : A \quad \neg\text{expansive}(e_1) \qquad \Sigma; \Delta \Leftarrow\!\!+ \{x : \mathrm{GEN}_A(\Sigma, \Delta)\} \vdash_R e_2 : B}{\Sigma; \Delta \vdash_R \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : B}$

$\text{expansive}(c) = \mathsf{false}$
$\text{expansive}(w) = \mathsf{false}$
$\text{expansive}(\lambda w.e) = \mathsf{false}$
$\text{expansive}(\mathsf{fix}\, f(w).\, e) = \mathsf{false}$
$\text{expansive}(e_1 e_2) = \mathsf{true}$
$\text{expansive}(\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2) = \text{expansive}(e_1) \vee \text{expansive}(e_2)$
$\text{expansive}(e \cdot w) = \text{expansive}(e)$
$\text{expansive}(\{\}) = \mathsf{false}$
$\text{expansive}(e_1 \,\mathsf{with}\, \{w = e_2\}) = \text{expansive}(e_1) \vee \text{expansive}(e_2)$
$\text{expansive}(\ell) = \mathsf{false}$
$\text{expansive}(\mathsf{ref}\; e) = \mathsf{true}$
$\text{expansive}(!e) = \text{expansive}(e)$
$\text{expansive}(e_1{:=}e_2) = \text{expansive}(e_1) \vee \text{expansive}(e_2)$

Figure 5.20: The new typing rules to handle references in the record calculus. These are standard [Har94, Wri95].

5.9.1 Adding References to the Staged and Record Calculi

The following syntax is added to the staged language. The resulting language is the same as λ^{open}_{poly} except open. The same syntax is added to the record calculus as well.

$$e \in Exp ::= \ldots \mid \ell \mid \mathsf{ref}\ e \mid !e \mid e{:=}e$$
$$\ell \in Location$$

The operational semantics of λ^{gen}_{poly} is extended with references as shown in Figure 5.17. The definitions of FV, FV^n and substitution are extended straightforwardly. An extension is also made to the staged type system as shown in Figure 5.18. This extension requires a new reference type and a store typing to be added to the judgments.

$$A \in SType ::= \ldots \mid A\ \mathsf{ref}$$
$$\Sigma \in SStoreTyping = Location \rightharpoonup SType$$

The store typing is used to look up the types of the locations occurring free (see the TSLOC rule). Let-bindings now have to take memory expansion into account when generalizing types. This is done by the $\mathrm{expansive}^n$ predicate in [KYC06], which is an adaptation of Wright's original definition [Wri95].

Definition 5.9.1. $\mathrm{expansive}^n(e)$ is as defined in [KYC06], except the following cases:

$$\begin{aligned}\mathrm{expansive}^n(\lambda x.e) &= \mathsf{false}\\ \mathrm{expansive}^n(\lambda^* x.e) &= \mathsf{false}\\ \mathrm{expansive}^n(\mathsf{fix}\, f(x).\, e) &= \mathsf{false}\end{aligned}$$

We modified the definition of expansiveness in [KYC06] because that definition is unnecessarily conservative for abstractions. Our definition is still safe, and preserves *demotion-closedness* because only stage-1 values can be demoted to stage-0, and stage-1 values do not contain holes not filled in yet. In other words, any hole that possibly exists under the abstraction has to be filled in before the lambda abstraction can be demoted to stage-0, making the abstraction non-expansive at any stage. The following expression, for example, is rejected by the λ^{open}_{poly} type system, but is succesfully accepted with the modification we gave.

$$\langle \mathsf{let}\, id = (\lambda x.\mathsf{let}\, t = \text{`}(\langle 0\rangle)\ \mathsf{in}\ x)\ \mathsf{in}\ id(1), id(\mathtt{true})\rangle$$

Later on, in Theorem 5.9.13, we will state that the record calculus provides a type system equal to λ^{open}_{poly}(with the modification given above). With the original definition of expansiveness we would be able to get only an $\Longrightarrow$ relation instead of $\Longleftrightarrow$.

The record calculus operational semantics and the type system are extended with references analogously as shown in Figures 5.19 and 5.20, respectively. The definition of possibly memory-expanding expressions is also given. In the typing rules omitted in Figure 5.20, the store typing is simply threaded through a proof tree.

Below we define safe-β-reductions: reductions that are guaranteed to not modify the store. This is used in the main theorem which states that translating a staged expression and then evaluating it in the record semantics produces the same side-effects as evaluation using the staged semantics.

Definition 5.9.2 (Side-effect freedom)**.** An expression e is said to be "side-effect-free", denoted as $SEF(e)$, if it is guaranteed not to change the store when evaluated. The formal definition is as follows:

$$\begin{aligned}
SEF(c) &= \text{true} \\
SEF(w) &= \text{true} \\
SEF(\lambda w.e) &= \text{true} \\
SEF(\mathsf{fix}\, f(x).\, e) &= \text{true} \\
SEF(e_1 e_2) &= \text{false} \\
SEF(\mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2) &= SEF(e_1) \wedge SEF(e_2) \\
SEF(e_1 \,\mathsf{with}\, \{a = e_2\}) &= SEF(e_1) \wedge SEF(e_2)
\end{aligned}
\qquad
\begin{aligned}
SEF(e \cdot a) &= SEF(e) \\
SEF(\{\}) &= \text{true} \\
SEF(\ell) &= \text{true} \\
SEF(\mathsf{ref}\ e) &= \text{false} \\
SEF(!e) &= SEF(e) \\
SEF(e_1 {:=} e_2) &= \text{false}
\end{aligned}$$

Definition 5.9.3 (Safe β-reduction)**.** The following are defined to be safe β-reductions.

$$\begin{aligned}
(\lambda w.e_1)e_2 &\longrightarrow_{|\beta|} e_1[w\backslash e_2] \text{ if } SEF(e_2) \\
\mathsf{let}\, w = e_1 \,\mathsf{in}\, e_2 &\longrightarrow_{|\beta|} e_2[w\backslash e_1] \text{ if } SEF(e_1) \\
(e_2 \,\mathsf{with}\, \{a_1 = e_1\})\cdot a_2 &\longrightarrow_{|\beta|} e_2 \cdot a_2 \text{ if } a_1 \neq a_2 \text{ and } SEF(e_1) \\
(e_2 \,\mathsf{with}\, \{a = e_1\})\cdot a &\longrightarrow_{|\beta|} e_1 \text{ if } SEF(e_2) \\
e \,\mathsf{with}\, \{a_1 = e_1\} \,\mathsf{with}\, \{a_2 = e_2\} &\longrightarrow_{|\beta|} e \,\mathsf{with}\, \{a_2 = e_2\} \,\mathsf{with}\, \{a_1 = e_1\} \\
&\qquad \text{if } a_1 \neq a_2, SEF(e_1) \text{ and } SEF(e_2) \\
e \,\mathsf{with}\, \{a = e_1\} \,\mathsf{with}\, \{a = e_2\} &\longrightarrow_{|\beta|} e \,\mathsf{with}\, \{a = e_2\} \text{ if } SEF(e_1)
\end{aligned}$$

5.9.2 Accounting for References in the Translation

We present a new version of the translation in Figure 5.21 that converts hole-filling into function application where holes become arguments. The example we gave at the beginning of the section, $\langle x + \grave{}(\mathsf{ref}\ 0; \langle 1 \rangle)\rangle$, for instance, translates to $(\lambda h.\lambda r_1.r_1 x + h(r_1))(\mathsf{ref}\ 0; \lambda r_1.1)$. Call-by-value semantics ensures that holes are evaluated before being filled in, preserving the order of evaluation. Below is the classical exponentiation example, written using a reference. Instead of threading the exponent value through recursive calls, we keep it as a global variable and decrement before each recursion. Even though the translation renames variables, not to harm readability of the code, we do not rename them unless they are accessed from a record. The given code generates the function $(\lambda x.x \times x \times x \times 1)$ which takes

$\llbracket c \rrbracket_{R_0,\ldots,R_n} = (c, \texttt{nil})$

$\llbracket x \rrbracket_{R_0,\ldots,R_n} = (R_n(x), \texttt{nil})$

$\llbracket \lambda x.e \rrbracket_{R_0,\ldots,R_n} = (\lambda z.e_0, H)$
  where $\llbracket e \rrbracket_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} = (e_0, H)$, and z is fresh.

$\llbracket \lambda^* x.e \rrbracket_{R_0,\ldots,R_n} = \llbracket \lambda z.[x^n \overset{n}{\mapsto} z]e \rrbracket_{R_0,\ldots,R_n}$, where z is fresh

$\llbracket \mathsf{fix}\, f(x).\, e \rrbracket_{R_0,\ldots,R_n} = (\mathsf{fix}\, g(z).\, e_0, H)$
  where $\llbracket e \rrbracket_{R_0,\ldots,R_n \,\mathsf{with}\, \{f=g, x=z\}} = (e_0, H)$, and g, z are fresh.

$\llbracket e_1 e_2 \rrbracket_{R_0,\ldots,R_n} = (e_1' e_2', zip(H_1, H_2))$
  where $\llbracket e_1 \rrbracket_{R_0,\ldots,R_n} = (e_1', H_1)$ and $\llbracket e_2 \rrbracket_{R_0,\ldots,R_n} = (e_2', H_2)$.

$\llbracket \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \rrbracket_{R_0,\ldots,R_n} = (\mathsf{let}\, z = e_1' \,\mathsf{in}\, e_2', zip(H_1, H_2))$
  where $\llbracket e_1 \rrbracket_{R_0,\ldots,R_n} = (e_1', H_1)$ and $\llbracket e_2 \rrbracket_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} = (e_2', H_2)$, and z is fresh.

$\llbracket \langle e \rangle \rrbracket_{R_0,\ldots,R_n} = ((\lambda \vec{\pi}.\lambda r.e_0)\vec{e_p}, H)$
  where $\llbracket e \rrbracket_{R_0,\ldots,R_n,r} = (e_0, \{(\vec{\pi}, \vec{e_p})\} :: H)$ and r is fresh.

$\llbracket \text{`}(e) \rrbracket_{R_0,\ldots,R_n,R_{n+1}} = (\pi(R_{n+1}), \{(\pi, e_0)\} :: H)$
  where $\llbracket e \rrbracket_{R_0,\ldots,R_n} = (e_0, H)$ and π is fresh.

$\llbracket \mathsf{run}(e) \rrbracket_{R_0,\ldots,R_n} = (e_0\{\}, H)$ where $\llbracket e \rrbracket_{R_0,\ldots,R_n} = (e_0, H)$.

$\llbracket \mathsf{lift}(e) \rrbracket_{R_0,\ldots,R_n} = (\mathsf{let}\, \pi = e_0 \,\mathsf{in}\, \lambda r.\pi, H)$ where $\llbracket e \rrbracket_{R_0,\ldots,R_n} = (e_0, H)$, and π is fresh.

$\llbracket \ell \rrbracket_{R_0,\ldots,R_n} = (\ell, \texttt{nil})$

$\llbracket \mathsf{ref}\ e \rrbracket_{R_0,\ldots,R_n} = (\mathsf{ref}\ e_0, H)$ where $\llbracket e \rrbracket_{R_0,\ldots,R_n} = (e_0, H)$.

$\llbracket !e \rrbracket_{R_0,\ldots,R_n} = (!e_0, H)$ where $\llbracket e \rrbracket_{R_0,\ldots,R_n} = (e_0, H)$.

$\llbracket e_1 {:=} e_2 \rrbracket_{R_0,\ldots,R_n} = (e_1' {:=} e_2', zip(H_1, H_2))$
  where $\llbracket e_1 \rrbracket_{R_0,\ldots,R_n} = (e_1', H_1)$ and $\llbracket e_2 \rrbracket_{R_0,\ldots,R_n} = (e_2', H_2)$.

The zip function is defined below, where :: is the cons operation:

$$\begin{array}{rcl} zip(h_1 :: H_1, h_1 :: H_2) & = & (h_1 \cup h_2) :: zip(H_1, H_2) \\ zip(\texttt{nil}, H_2) & = & H_2 \\ zip(H_1, \texttt{nil}) & = & H_1 \end{array}$$

Figure 5.21: Transformation modified to handle expressions with side-effects.

the cube of its argument. The translation returns the function

$$\lambda y.(\lambda r.r{\cdot}x \times (\lambda r.r{\cdot}x \times (\lambda r.r{\cdot}x \times (\lambda r.1)r)r)r)\{x = y\}$$

which does the same thing through record operations.

$$\begin{array}{l}
[\![\mathsf{let}\, n = \mathsf{ref}\ 0\, \mathsf{in} \\
\ \mathsf{let}\, pow = \mathsf{fix}\, gen().\, \mathsf{if}\ !n = 0\ \mathsf{then}\ \langle 1\rangle\ \ \mathsf{else}\ \langle x \times \text{`}(n{:=}!n - 1; gen())\rangle \\
\ \mathsf{in}\ n{:=}3; \mathsf{run}\ \langle \lambda x.\text{`}(pow())\rangle]\!]^0 =
\end{array}$$

$$\begin{array}{l}
\mathsf{let}\, n = \mathsf{ref}\ 0\, \mathsf{in} \\
\mathsf{let}\, pow = \mathsf{fix}\, gen().\, \mathsf{if}\ !n = 0\ \mathsf{then}\ (\lambda r.1)\ \ \mathsf{else}\ (\lambda \pi.\lambda r.r{\cdot}x \times \pi(r))(n{:=}!n - 1; gen()) \\
\mathsf{in}\ n{:=}3; (\lambda \pi.\lambda r.\lambda y.\pi(r\ \mathsf{with}\ \{x = y\}))(pow())\{\}
\end{array}$$

The power function below is yet another version that counts the number of multiplications generated. It is adapted from [KKcS08, §6].

$$\begin{array}{l}
[\![\mathsf{let}\, cnt = \mathsf{ref}\ 0\, \mathsf{in} \\
\ \mathsf{let}\, pow = \mathsf{fix}\, gen(n).\, \lambda x.\mathsf{if}\ n = 0\ \mathsf{then}\ \langle 1\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ cnt{:=}!cnt + 1;\ \langle \text{`}(x) \times \text{`}(gen\ n\ x)\rangle \\
\ \mathsf{in}\ \mathsf{run}\ \langle \lambda x.\text{`}(pow\ 3\ \langle x\rangle)\rangle]\!]^0 =
\end{array}$$

$$\begin{array}{l}
\mathsf{let}\, cnt = \mathsf{ref}\ 0\, \mathsf{in} \\
\mathsf{let}\, pow = \mathsf{fix}\, gen(n).\, \lambda x.\mathsf{if}\ n = 0\ \mathsf{then}\ (\lambda r.1) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ cnt{:=}!cnt + 1;\ (\lambda \pi_1.\lambda \pi_2.\lambda r.\pi_1(r) \times \pi_2(r))\ x\ (gen\ n\ x) \\
\mathsf{in}\ (\lambda \pi.\lambda r.\lambda y.\pi(r\ \mathsf{with}\ \{x = y\}))(pow\ 3\ (\lambda r.r{\cdot}x))\{\}
\end{array}$$

The example below produces a specialized version of vector product. For this example we assume that there is a built-in function nth that, given i and a list ℓ, returns the i^{th} element of ℓ. This is a two-level specialization. The first level produces a generator specialized for a fixed length; the second level specializes the code further for the values kept in a vector. For instance, $\mathsf{run}(prod\ 2)$ gives

$$\lambda v.\langle \mathsf{nth}\ \text{`}(\mathsf{lift}(2))\ w \times \text{`}(\mathsf{lift}(\mathsf{nth}\ 2\ v)) + \mathsf{nth}\ \text{`}(\mathsf{lift}(1))\ w \times \text{`}(\mathsf{lift}(\mathsf{nth}\ 1\ v)) + \text{`}(\langle 0\rangle)\rangle$$

and $\langle \lambda w.\text{`}(\mathsf{run}(prod\ 2)[5;7])\rangle$ is $\langle \lambda w.(\mathsf{nth}\ 2\ w) \times 5 + (\mathsf{nth}\ 1\ w) \times 7 + 0\rangle$. We can now run this code value and apply it to a vector of length 2, such as $[2;3]$.

$$\begin{array}{l}
[\![\mathsf{let}\, prod = \\
\quad \mathsf{let}\, aux = \mathsf{fix}\, gen(n). \\
\qquad \mathsf{if}\ n = 0\ \mathsf{then}\ \langle\langle 0\rangle\rangle \\
\qquad \mathsf{else}\ \langle\langle \mathsf{nth}\ \text{`}(\mathsf{lift}(\text{`}(\mathsf{lift}(n))))\ w \times \text{`}(\mathsf{lift}(\mathsf{nth}\ \text{`}(\mathsf{lift}(n))\ v)) + \text{`}(\text{`}(gen(n-1)))\rangle\rangle \\
\quad \mathsf{in}\ \lambda n.\langle \lambda v.\text{`}(aux\ n)\rangle \\
\ \mathsf{in}\ (\mathsf{run}\ \langle \lambda w.\text{`}(\mathsf{run}(prod\ 2)[5;7])\rangle)[2;3]]\!]^0 =
\end{array}$$

$$
\begin{array}{l}
\textsf{let}\ prod = \\
\quad \textsf{let}\ aux = \textsf{fix}\ gen(n). \\
\qquad\quad \textsf{if}\ n = 0\ \textsf{then}\ \lambda r_2.\lambda r_1.0 \\
\qquad\quad \textsf{else}\ (\lambda\pi_4.\lambda\pi_5.\lambda\pi_6.\lambda r_2.\ (\lambda\pi_1.\lambda\pi_2.\lambda\pi_3.\lambda r_1.(\textsf{nth}\ (\pi_1 r_1)\ (r_1{\cdot}w)) \times \pi_2(r_2) + \pi_3(r_2)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textsf{let}\ \pi = \pi_4(r_2)\ \textsf{in}\ \lambda r.\pi) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textsf{let}\ \pi = \textsf{nth}\ (\pi_5\, r_2)\ (r_2{\cdot}v)\ \textsf{in}\ \lambda r.\pi) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\pi_6(r_2))) \\
\qquad\qquad\quad (\textsf{let}\ \pi = n\ \textsf{in}\ \lambda r.\pi) \\
\qquad\qquad\quad (\textsf{let}\ \pi = n\ \textsf{in}\ \lambda r.\pi) \\
\qquad\qquad\quad (gen(n-1)) \\
\quad \textsf{in}\ \lambda n.(\lambda\pi_7.\lambda r.\lambda v'.\pi_7(r\ \textsf{with}\ \{v = v'\}))(aux\ n) \\
\textsf{in}\ (\lambda\pi_8.\lambda r.\lambda w'.\pi_8(r\ \textsf{with}\ \{w = w'\}))((prod\ 2)\{\}[5;7])\{\}[2;3]
\end{array}
$$

The value that is returned by the transformation of an expression e now has the form of a pair: $(e_0,\ \{(\pi_i, e_i)\}_1^m :: \ldots :: \{(\pi_i, e_i)\}_1^p)$. Here, e_0 is the actual result of the transformation where each hole not enclosed by a quotation has been replaced by a unique variable π. The second item in the return value, a list of variable and expression sets, contains these unique hole-filler variables accompanied with the corresponding antiquoted expression. A set in the returned list corresponds to a specific stage. The stage gets closer to 0 as we move from left to right in the list. The transformation of a quotation retrieves the variable-expression pairs from the top of the list, and uses them to "fill" in the holes via function application. For the example given above, $[\![\,\grave{}(\textsf{ref}\ 0; \langle 1\rangle)]\!]_{\{\},r_1}$ gives $(\pi(r_1), [\{(\pi, \textsf{ref}\ 0; \lambda r.1)\}])$. Based on this value, $[\![\langle x + \grave{}(\textsf{ref}\ 0; \langle 1\rangle)\rangle]\!]_{\{\}}$ returns $((\lambda\pi.\lambda r_1.r_1{\cdot}x + \pi(r_1))(\textsf{ref}\ 0; \lambda r.1), \texttt{nil})$. Note that the number of sets returned is equal to the depth of the transformed expression:

Lemma 5.9.4. *Let e be a stage-n program, and $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$. The length of H is equal to the depth of e, giving $n \geq$ length of H.*

Proof. By a straightforward induction on the structure of e. Note that length of $zip(H_1, H_2)$ is equal to max(length of H_1, length of H_2). The only expression that adds a new item to H is quotation, and the only expression that removes an item from H is antiquotation. □

Translation of types and judgments stays the same. There are two extensions to be made. The first one converts a staged store typing to a record store typing:

$$[\![\{\ell_i : A\}]\!] = \{\ell_i : [\![A]\!]\}$$

The second extension translates a store:

Definition 5.9.5 (Store translation)**.** Let $\mathcal{S} = \{\ell_i : v_i\}$ be a λ^{open}_{poly} store. Its translation to a λ^{rec}_{poly} store is defined as follows:

$$[\![\{\ell_i : v_i\}]\!] = \{\ell_i : v'_i\} \text{ where } (v'_i, \texttt{nil}) = [\![v_i]\!]_{\{\}}$$

Note that all the values in $\mathcal{S}$ are stage-0 values, and the second item of the result of translating a stage-0 is guaranteed to always be `nil` by Lemma 5.9.4.

5.9.3 Relating the Staged and Record Calculi

We now show that the translation preserves the order of evaluation and the record calculus still provides a sound type system with respect to staged semantics with side-effects. The properties related to the record calculus are still valid with the extension made. We do not repeat them. We first give auxiliary definitions and lemmas. The $Close$ operation below packs the result of a translation into a single function application. We use the notation $\{(\vec{\pi}, \vec{e})\}$ for the set $\{(\pi_i, e_i)\}_1^m$.

Definition 5.9.6. Let $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$. $Close$ is defined as

$$Close([\![e]\!]_{R_0,\ldots,R_n}) = (\lambda\vec{\pi_m}.(\cdots((\lambda\vec{\pi_1}.e_0)\vec{e_1})\cdots))\vec{e_m}$$

Below is the theorem stating that record operational semantics together with the translation is equivalent to staged operational semantics. The crucial property expressed by this theorem is that translation from the staged language into the record calculus preserves the order of evaluation, and hence the side effects of an expression.

Theorem 5.9.7. *Let e_1 be a stage-n λ^{open}_{poly} expression such that $FV^n(e_1) = \varnothing$. If $\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_2$, then $[\![\mathcal{S}]\!], Close([\![e_1]\!]_{\{\},R_1,\ldots,R_n}) \ \longrightarrow_R \ [\![\mathcal{S}']\!], e_2'$ such that $e_2' \longrightarrow^*_{|\beta|} Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n})$.*

Proof. By structural induction on e_1, based on the last applied reduction rule. As a remark, an examination of the proof shows that safe-β reductions taken above are only "administrative" reductions in the style of Danvy and Filinski [DF92]. $\square$

The following definition states the consistency between stores and store typings.

Definition 5.9.8 (Well-typed stores)**.** A store $\mathcal{S}$ is well typed with respect to a store typing Σ, denoted $\Sigma \models \mathcal{S}$, if and only if $dom(\mathcal{S}) = dom(\Sigma)$, and $\Sigma; \varnothing \vdash_R \mathcal{S}(\ell) : \Sigma(\ell)$ for any $\ell \in dom(\mathcal{S})$.

Theorem 5.9.9 (Preservation)**.** *Let e_1 be a stage-n λ^{open}_{poly} expression. If*

$$\Sigma; \Delta \vdash_R Close([\![e_1]\!]_{\{\},R_1,\ldots,R_n}) : A \text{ and } \mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_2$$

such that $\Sigma \models [\![\mathcal{S}]\!]$, then for some $\Sigma' \supseteq \Sigma$ we have

$$\Sigma'; \Delta \vdash_R Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n}) : A \text{ and } \Sigma' \models [\![\mathcal{S}']\!]$$

Proof. By Theorem 5.9.7 we have $[\![\mathcal{S}]\!], Close([\![e_1]\!]_{\{\},R_1,\ldots,R_n}) \ \longrightarrow_R \ [\![\mathcal{S}']\!], e_2'$ such that $e_2' \longrightarrow^*_{|\beta|}$ $Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n})$. By the preservation property of the record calculus, there exists Σ''

such that $\Sigma'' \supseteq \Sigma$ and $\Sigma'' \models [\![\mathcal{S}']\!]$ giving the judgment $\Sigma''; \Delta \vdash_R e_2' : A$. Using the preservation property of the record calculus again, and the fact that $e_2' \longrightarrow^*_{|\beta|} Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n})$ does not modify the store, we get $\Sigma''; \Delta \vdash_R Close([\![e_2]\!]_{\{\},R_1,\ldots,R_n}) : A$. $\square$

Theorem 5.9.10 (Progress). *Let e_1 be a stage-n λ^{open}_{poly} expression. If $\Sigma; \Delta \vdash_R Close([\![e_1]\!]_{\{\},R_1,\ldots,R_n}) : A$, then either $e_1 \in Val^n$, or for any store $\mathcal{S}$ such that $\Sigma \models \mathcal{S}$, there exist e_2 and $\mathcal{S}'$ such that $\mathcal{S}, e_1 \longrightarrow_n \mathcal{S}', e_2$.*

Proof. By structural induction on e_1. $\square$

Theorem 5.9.11 (Soundness). *Let e_1 be a stage-0 λ^{open}_{poly} expression and $[\![e_1]\!]_{\{\}} = (e_0, \mathtt{nil})$. If $\varnothing; \varnothing \vdash_R e_0 : A$, then either $e_1 \Uparrow$, or there exists $e_2 \in Val^0$ such that $[\![e_2]\!]_{\{\}} = (e_0', \mathtt{nil})$ and $\varnothing, e_1 \longrightarrow^*_0 \mathcal{S}, e_2$ and $\Sigma; \varnothing \vdash_R e_0' : A$ where $\Sigma \models \mathcal{S}$.*

Proof. Follows from Theorems 5.9.9 and 5.9.10. $\square$

We have the following relation between expansion tests of record calculus and staged typing:

Lemma 5.9.12. *For any stage-n expression e, $\mathrm{expansive}^n(e) \iff \mathrm{expansive}(e_0)$ where $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, H)$.*

Proof. By a straightforward induction on the structure of e. $\square$

Theorem 5.6.6, which stated that the record type system combined with the translation is equal to the λ^{open}_{poly} type system, is now stated as follows:

Theorem 5.9.13. *Let e be a staged program.*

$$\Sigma; \Delta_0, \ldots, \Delta_n \vdash_S e : A \iff [\![\Sigma]\!]; [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R Close([\![e]\!]_{R_0,\ldots,R_n}) : [\![A]\!]$$

Proof. By induction on the structure of e. $\square$

5.9.4 Handling Pluggable Declarations in the Presence of References

In Section 5.8 we extended the staged language with pluggable declarations. We also gave a translation to the record calculus. In the presence of references, the translation has to be modified in the same way we did for the core language. The new definition of the translation is given below.

$[\![\langle\rangle]\!]_{R_0,\ldots,R_n} = (\lambda\kappa.\lambda y.\lambda r.\, y(r),\ \mathtt{nil})$

$[\![\langle x = e\rangle]\!]_{R_0,\ldots,R_n} = ((\lambda\vec{\pi}.\lambda\kappa.\lambda y.\lambda r.\mathsf{let}\ z = e_0\ \mathsf{in}\ y(r\ \mathsf{with}\ \{x = z\}))\vec{e_p}, H)$
where $[\![e]\!]_{R_0,\ldots,R_n,r} = (e_0, \{(\vec{\pi}, \vec{e_p})\} :: H)$, and r, z are fresh.

$[\![\mathsf{let}\ `(e_1)\ \mathsf{in}\ e_2]\!]_{R_0,\ldots,R_n,R_{n+1}} = (\pi\,\kappa\,(\lambda r.e_0')R_{n+1},\ \ zip(\{(\pi, e_0)\} :: H,\ H'))$
where $[\![e_1]\!]_{R_0,\ldots,R_n} = (e_0, H)$, π, r are fresh, and $[\![e_2]\!]_{R_0,\ldots,R_n,r} = (e_0', H')$

A similar modification to the desugaring function is also needed. Because this change is along the same lines of the change made to the translation function, we omit it.

5.10 Related Work

We now compare the papers that are closest to our work in this chapter.

Translating a staged language to record calculus, as motivated in Section 5.1, has previously been proposed by Kameyama, Kiselyov and Shan [KKcS08]. They translate $\lambda^{\alpha}_{1\nu}$, a two-stage version of Taha and Nielsen's λ^{α} [TN03], to System F [Gir72, Rey74] with tuples and higher order polymorphism. Our work differs from [KKcS08] as follows.

- Our translation is not restricted to two-stage program generation; it is multi-staged.
- Targeting "PG by program construction", our source language allows freely-open fragments, as opposed to the "PG by partial evaluation" approach in [KKcS08] which rejects fragments containing free variables that are not in the scope of an outer binding.
- The translation of $\lambda^{\alpha}_{1\nu}$ is guided by type and environment classifier annotations. Neither the source nor the target language in our translation contains type annotations. The target language, record calculus, already has a principal type inference algorithm defined. We simply use this algorithm to infer types.
- We provide a proof of the equivalence of the dynamic semantics of staged computation and record calculus. For a similar relation, [KKcS08] gives a conjecture.
- We have let-polymorphism (i.e. rank-1 polymorphism) as in λ^{open}_{poly}. We do not allow higher order polymorphism as in $\lambda^{\alpha}_{1\nu}$. This prevents polymorphic types to live across stages. The following program (we omitted type annotations) is typable in MetaML-like typing (e.g. in $\lambda^{\alpha}_{1\nu}$) but not in our system.

$$\langle \mathsf{let}\ f = \lambda x.x\ \mathsf{in}\ \grave{}(\langle f(1), f(\mathtt{true})\rangle)\rangle$$

An interesting question is how the idea of translation would apply to a PG by partial evaluation language with rank-1 polymorphism and no type annotations, such as MetaML [TS00]. The example above requires cross-stage persistence of polymorphic types. Our translation fails to type-check it because the translation converts quotations to lambda abstraction and antiquotations to function applications, and function parameters cannot have polymorphic types in rank-1 polymorphism. (Recall that Kameyama et al. assume higher rank polymorphism and the existence of type annotations, so they do not face this problem.) To have variable bindings to be available cross-stage, we could define a new translation which does not throw away the topmost environment when there is an antiquotation, but puts it aside to reuse when a quotation is encountered later on (instead of starting with a fresh environment), so that previously defined variables will be available. The example above would then translate to

$$\lambda r.\mathsf{let}\, g = \lambda x.x \,\mathsf{in}\, (\lambda r.g(1), g(\mathtt{true}))(r \,\mathsf{with}\, \{f = g\})$$

which successfully type-checks. This translation, however, suffers from *scope extrusion*. Consider $\langle \lambda x.\grave{}(\mathsf{run}(\langle x \rangle))\rangle$, which should be rejected because a fragment with a free variable is being run. Its translation would be $\lambda r.\lambda y.(\lambda r.y)\{\}(r \,\mathsf{with}\, \{x = y\})$, which is admissible by the record type system. We plan to investigate this problem as a future research.

Chen and Xi [CX03] give a translation to convert fragments to first-order abstract syntax expressions. They represent program variables using deBruijn indices. Their target language is second-order lambda calculus with recursion. One advantage is that they can translate first-order abstract syntax (without holes) back to regular syntax (with quotations). A problem in their system is "at level $k > 0$, a bound variable merely represents a deBruijn index and a binding may vanish or occur 'unexpectedly'" [CX03]. An example that illustrates this problem in the existence of references is given by Kim, Yi and Calcagno [KYC06, §6.4]. In [CX03], polymorphism is restricted to stage 0 only.

Kim, Yi and Calcagno define a language called λ^{open}_{poly} that allows program generation using freely-open code fragments [KYC06]. The language combines many features such as references, variable hygiene to avoid unintended capture as well as intentional variable capturing, and let-polymorphism together with core program generation facilities of quotation, antiquotation, and "run" (to execute quoted fragments). A sound type system and a principal type inference algorithm are provided. We took λ^{open}_{poly} as a starting point for a staged language.

Nanevski [Nan02] takes the approach of relaxing Davies' notion of closed code [DP96, Dav96] to allow manipulation of open code together with a sound "run" construct. He introduces a new semantic category, *names*, that stands for the free variables in a code piece. Free variables become part of a fragment's type as the "support set" — the variables that the code piece depends on. Only code values that have empty support sets can be "run"; other code values can only be used to fill in holes. Similar to row-polymorphism, there exists support set polymorphism. Pattern matching for code values is introduced. This makes it possible to do computation based on the structure of a fragment. An example is given that performs β-reduction inside quotations. An important feature of the type system is subtyping: the support set of a code value can be loosened to allow its use in different contexts. Because no type information is kept for free variables inside a support set, only the existence or absence of a variable can be expressed. In our work, we use subtyping constraints as defined by Pottier [Pot00b]. Subtyping constraints subsume Nanevski's notion of subtyping and provide more expressibility. In particular, they successfully address the subtyping requirement revealed by the library specialization problem. The definition of subtyping in [Nan02] does not suffice for this problem. To our knowledge, a staged type system with subtyping constraints is new.

In addition to these differences, we added *pluggable declarations* to our staged language and showed that pluggable declarations are a syntactic sugaring that helps us avoid higher order functions.

5.11 Conclusions

Guaranteeing that a program generator produces type-safe programs has received extensive interest in the literature. We have tackled the same problem in the context of PG by program construction. We have shown that the problem reduces to type-checking in record calculus, which is an extensively studied area. This allows us to apply formal properties of the record calculus to program generation. An example is subtyping; we have discussed how a staged type system can be enriched with subtyping constraints. The close relation between the record and staged calculi exists in the presence of side-effecting expressions as well. We have also shown how to extend the language and the type system with pluggable declarations. These extensions yield a powerful type system that can successfully address the important requirements motivated by the library specialization problem.

We believe that the three extensions we discussed, namely subtyping, pluggable declarations, and updatable references, are orthogonal; it is possible to combine them without any fundamental problems. None of the extensions, nor the core program generation language, requires the programmer to write any type annotations; existing type-inference algorithms can be used to infer the types. These features form a nice package as a program generation system.

Chapter 6

Conclusions and Future Work

Program generation (PG) is a widely applicable technique. It poses two important challenges for the developers of program generation systems. The first challenge is the efficiency of generation: How can we generate code faster? The second is the safety of the generated program: How can we ensure that the generated code will be type-safe? In this dissertation, we focused on these two problems in the context of "PG by program construction".

There are two approaches that can be taken to address the first challenge.

- *Avoid program generation if it is not likely to bring advantage.* To do this, we experimented with the idea of "just-in-time" generation (Chapter 2). Our empirical results show that this is an effective technique that improves the efficiency gains from program generation while imposing negligible overhead.

- *Take advantage of available fragments of the program that will be generated to reduce the cost of generation.* We discussed two techniques that are based on this idea. The first technique uses source-level transformations to optimize the intermediate representation of a fragment (Chapter 3). This technique builds on the principle of compositionality of the compiler for program generation. The second technique performs at compile time a portion of analyses that have to be done during runtime generation, so that less time will be spent for generation (Chapter 4). To do this, we defined a framework that leads to staging of analyses. We also provided benchmarking results for both techniques to show that the techniques are indeed useful.

To address the second challenge of program generation, we showed that record calculus can be used to develop a powerful type system (Chapter 5). We gave a translation to convert program generators to record calculus programs, and proved that the two programs execute equivalently. This makes it possible to use a record calculus type system to type-check program generators. We showed that this type system can further be extended with more advanced features, such as subtyping, by using already-existing definitions and properties from the record calculus domain.

Below we discuss some ideas, which, we think, form interesting topics for future research.

In Chapter 2 we discussed how a program generator is written starting from non-generative code. It would be interesting to see how the same style applies to other applications that share similar characteristics with marshalling — applications where a generic algorithm handles many different kinds of data. It would also be interesting to see the effect of just-in-time generation on other applications.

In Chapter 3 we discussed the lessons we learned from our experience in applying source-level transformations on Java. The dynamic character of Java makes it difficult to optimize away many computations. Performing the transformations on a more static language, like C, may yield better results. Another possibility is to try a language whose compiler would require only one pass over a program. This improves our abilities for optimization as more information becomes available. In a prototype language we had achieved much higher speedup [Akt05, AK05].

In Chapter 4 we gave a framework for data-flow analysis that can be naturally staged. It would be interesting to see if flow/context insensitive analyses can be staged as well. Some variants of pointer and shape analysis are examples. Another question that is worth investigating is whether there is a systematic way to obtain representations from the traditional definition of analyses. Finally, using the analysis framework for a purpose different from staging is a possibility. Because fragments can be individually analyzed and reduced to representations using our framework, it may be possible to parallelize data-flow analysis of programs. There already exists work focusing on this problem [LRF95, LRM91, LR94, KGS94]. We believe that our framework has a potential for better results in this area as it can handle arbitrarily small fragments with multiple exits.

In Chapter 5 we gave theoretical results. The program generation language we used is an ML-like functional language. It would be interesting to see how the results we found could be extended to a more mainstream, object-oriented language. Because we are translating program generators to programs in record calculus, and because records form the fundamentals of object theory [AC96], we believe there is a high potential in the generalization of our translation to object oriented languages. Inheritance, a key feature in object-orientation, may pose a difficulty. We expect theoretical and practical results obtained in the areas of subtyping in records [Pot00b] and type inference of objects [Pal95, PWO97, PZ02] to be very useful in overcoming the challenges of the extension to object-orientation. Lastly, we think that implementation of a powerful type system for a staged version of a mainstream language is an important goal that the program generation community should achieve.

According to type theorists, data-flow analysis is a kind of a type system; according to compiler developers, type-checking is a kind of a data-flow analysis. This dual view shows the close relation between type-checking and data-flow analysis. A question that is worth investigating is whether a staged type system can be generalized as a framework for data-flow analysis of staged programs. Another related idea is whether it is possible to optimize the program that will be generated, by statically optimizing the translation of the generator in the record calculus using traditional compiler optimizations. This is sensible because the execution of a staged program and its translation are equivalent.

We have stated that a staged type system can be obtained by translating a staged program to the record calculus and using a record type system. A question that naturally arises is how to design a standalone staged type system that is as powerful (e.g. has subtyping constraints, references, pluggable declarations), but that does not require the use of a translation and dependence on a record type system. Deriving such a type system may be simply a matter of reverse engineering from the results we have given. We plan to investigate this problem in the near future.

We hope that the results shown in this dissertation will be useful to researchers in design and implementation of new program generation systems, and will lead to new research topics.

Appendix A

Proofs

A.1 Proofs of Theorems in Chapter 4

Proof of Theorem 4.1.4. The proof is by induction on the structure of P.

- Case 1. $P = \mathsf{skip}$

$$\begin{aligned}
\mathbf{abs}(\mathcal{F}^R[\![\mathsf{skip}]\!]r) &= \mathbf{abs}(id^R(r)) \\
&= \mathbf{abs}(r) \\
&= \mathbf{abs}(r) \ ; \ id \\
&= \mathbf{abs}(r) \ ; \ \mathbf{abs}(id_R) \\
&= \mathbf{abs}(r) \ ; \ \mathbf{abs}(\mathcal{R}[\![\mathsf{skip}]\!]) \\
&= \mathbf{abs}(r \ ;_R \mathcal{R}[\![\mathsf{skip}]\!])
\end{aligned}$$

- Case 2. $P = x = e$

$$\begin{aligned}
\mathbf{abs}(\mathcal{F}^R[\![x = e]\!]r) &= \mathbf{abs}(asgn^R(x, e)(r)) \\
&= \mathbf{abs}(r \ ;_R asgn_R(x, e)) \\
&= \mathbf{abs}(r \ ;_R \mathcal{R}[\![x = e]\!])
\end{aligned}$$

- Case 3. $P = P_1; P_2$
  By the induction hypothesis, we have, $\forall r \in R$,

$$\mathbf{abs}(\mathcal{F}^R[\![P_1]\!]r) = \mathbf{abs}(r \ ;_R \mathcal{R}[\![P_1]\!])$$
$$\mathbf{abs}(\mathcal{F}^R[\![P_2]\!]r) = \mathbf{abs}(r \ ;_R \mathcal{R}[\![P_2]\!])$$

  Now we work on $P_1; P_2$:

$$\begin{aligned}
\textbf{abs}(\mathcal{F}^R[\![P_1;P_2]\!]r) &= \textbf{abs}(\mathcal{F}^R[\![P_2]\!](\mathcal{F}^R[\![P_1]\!]r)) \\
&= \textbf{abs}((\mathcal{F}^R[\![P_1]\!]r) \;;_R \mathcal{R}[\![P_2]\!]) \qquad (A.1) \\
&= \textbf{abs}((\mathcal{F}^R[\![P_1]\!]r)); \textbf{abs}(\mathcal{R}[\![P_2]\!]) \\
&= \textbf{abs}(r \;;_R \mathcal{R}[\![P_1]\!]); \textbf{abs}(\mathcal{R}[\![P_2]\!]) \qquad (A.2) \\
&= \textbf{abs}(r); \textbf{abs}(\mathcal{R}[\![P_1]\!]); \textbf{abs}(\mathcal{R}[\![P_2]\!]) \\
&= \textbf{abs}(r); \textbf{abs}(\mathcal{R}[\![P_1]\!] \;;_R \mathcal{R}[\![P_2]\!]) \\
&= \textbf{abs}(r); \textbf{abs}(\mathcal{R}[\![P_1;P_2]\!]) \\
&= \textbf{abs}(r \;;_R \mathcal{R}[\![P_1;P_2]\!])
\end{aligned}$$

Induction hypothesis is used to derive (A.1) and (A.2).

- Case 4. $P = \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$
  By the induction hypothesis, we have, $\forall r \in R$,

$$\begin{aligned}
\textbf{abs}(\mathcal{F}^R[\![P_1]\!]r) = \textbf{abs}(r \;;_R \mathcal{R}[\![P_1]\!]) \\
\textbf{abs}(\mathcal{F}^R[\![P_2]\!]r) = \textbf{abs}(r \;;_R \mathcal{R}[\![P_2]\!])
\end{aligned}$$

Now we work on $\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$:

$$\begin{aligned}
&\textbf{abs}(\mathcal{F}^R[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]r) = \textbf{abs}((exp^R(e)\ ;\ (\mathcal{F}^R[\![P_1]\!] \wedge^R \mathcal{F}^R[\![P_2]\!]))r) \\
&= \textbf{abs}((\mathcal{F}^R[\![P_1]\!] \wedge^R \mathcal{F}^R[\![P_2]\!])(r \;;_R exp_R(e))) \\
&= \textbf{abs}(\mathcal{F}^R[\![P_1]\!](r \;;_R exp_R(e)) \wedge_R \mathcal{F}^R[\![P_2]\!](r \;;_R exp_R(e))) \\
&= \textbf{abs}(\mathcal{F}^R[\![P_1]\!](r \;;_R exp_R(e))) \wedge \textbf{abs}(\mathcal{F}^R[\![P_2]\!](r \;;_R exp_R(e))) \\
&= \textbf{abs}((r \;;_R exp_R(e)) \;;_R \mathcal{R}[\![P_1]\!]) \wedge \textbf{abs}((r \;;_R exp_R(e)) \;;_R \mathcal{R}[\![P_2]\!]) \qquad (A.3) \\
&= \textbf{abs}(r \;;_R exp_R(e)); \textbf{abs}(\mathcal{R}[\![P_1]\!]) \ \wedge\ \textbf{abs}(r \;;_R exp_R(e)); \textbf{abs}(\mathcal{R}[\![P_2]\!]) \\
&= \textbf{abs}(r); \textbf{abs}(exp_R(e)); \textbf{abs}(\mathcal{R}[\![P_1]\!]) \ \wedge\ \textbf{abs}(r); \textbf{abs}(exp_R(e)); \textbf{abs}(\mathcal{R}[\![P_2]\!]) \\
&= \textbf{abs}(r); \textbf{abs}(exp_R(e)); (\textbf{abs}(\mathcal{R}[\![P_1]\!]) \ \wedge\ \textbf{abs}(\mathcal{R}[\![P_2]\!])) \\
&= \textbf{abs}(r); \textbf{abs}(exp_R(e)); (\textbf{abs}(\mathcal{R}[\![P_1]\!] \wedge_R \mathcal{R}[\![P_2]\!])) \\
&= \textbf{abs}(r); \textbf{abs}(exp_R(e) \;;_R (\mathcal{R}[\![P_1]\!] \wedge_R \mathcal{R}[\![P_2]\!])) \\
&= \textbf{abs}(r); \textbf{abs}(\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]) \\
&= \textbf{abs}(r \;;_R (\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]))
\end{aligned}$$

Induction hypothesis is used to derive (A.3).

□

Proof of Theorem 4.1.6. The proof is by induction on the structure of P.

- Case 1. $P = \mathsf{skip}$

$$\begin{array}{rcl}
\mathbf{abs}_E(\mathcal{R}[\![\mathsf{skip}]\!]) & = & \mathbf{abs}_E((\top_{Env_R},\ id_R)) \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\top_{Env_R}(\ell))d',\ \mathbf{abs}(id_R)d') \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\top_R)d',\ id(d')) \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge (\lambda d.\top_{Data})d',\ d') \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \top_{Data},\ d') \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell),\ d') \\
& = & \lambda(\eta', d').(\eta',\ d') \\
& = & \mathcal{F}[\![\mathsf{skip}]\!]
\end{array}$$

- Case 2. $P = x = e$

$$\begin{array}{rcl}
\mathbf{abs}_E(\mathcal{R}[\![x = e]\!]) & = & \mathbf{abs}_E((\top_{Env_R},\ asgn_R(x,e))) \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\top_{Env_R}(\ell))d',\ \mathbf{abs}(asgn_R(x,e))d') \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \mathbf{abs}(\top_R)d',\ asgn(x,e)(d')) \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge (\lambda d.\top_{Data})d',\ asgn(x,e)(d')) \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell) \wedge \top_{Data},\ asgn(x,e)(d')) \\
& = & \lambda(\eta', d').(\lambda\ell.\eta'(\ell),\ asgn(x,e)(d')) \\
& = & \lambda(\eta', d').(\eta',\ asgn(x,e)(d')) \\
& = & \mathcal{F}[\![x = e]\!]
\end{array}$$

- Case 3. $P = \mathsf{break}\ \ell$

$$\begin{array}{rcl}
\mathbf{abs}_E(\mathcal{R}[\![\mathsf{break}\ \ell]\!]) & = & \mathbf{abs}_E((\top_{Env_R}[\ell \mapsto id_R], \top_R)) \\
& = & \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\top_{Env_R}[\ell \mapsto id_R](\ell'))d',\ \mathbf{abs}(\top_R)d') \\
& = & \lambda(\eta', d').(\Big(\lambda\ell'.\begin{cases} \eta'(\ell) \wedge \mathbf{abs}(id_R)d' & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \mathbf{abs}(\top_R)d' & \text{if } \ell \neq \ell' \end{cases}\Big), \top_{Data}) \\
& = & \lambda(\eta', d').(\Big(\lambda\ell'.\begin{cases} \eta'(\ell) \wedge d' & \text{if } \ell = \ell' \\ \eta'(\ell') & \text{if } \ell \neq \ell' \end{cases}\Big), \top_{Data}) \\
& = & \lambda(\eta', d').(\eta'[\ell \mapsto \eta'(\ell) \wedge d'],\ \top_{Data}) \\
& = & \mathcal{F}[\![\mathsf{break}\ \ell]\!]
\end{array}$$

- Case 4. $P = \ell : P'$

Let $(\eta, r) = \mathcal{R}[\![P']\!]$. By the induction hypothesis we have

$$\begin{aligned}
\mathcal{F}[\![P']\!] &= \textbf{abs}_E(\mathcal{R}[\![P']\!]) \\
&= \textbf{abs}_E((\eta, r)) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d',\ \textbf{abs}(r)d')
\end{aligned}$$

Now we work on $\textbf{abs}_E(\mathcal{R}[\![\ell : P']\!])$. Note that because we require all the programs to be legal, the incoming environment has ℓ mapped to $\top_{Data}$.

$$\begin{aligned}
&\textbf{abs}_E(\mathcal{R}[\![\ell : P']\!]) = \textbf{abs}_E((\eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta[\ell \mapsto \top_R](\ell'))d',\ \textbf{abs}(r \wedge_R \eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'. \begin{cases} \eta'(\ell) \wedge \textbf{abs}(\top_R)d' & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell' \end{cases},\ \textbf{abs}(r \wedge_R \eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'. \begin{cases} \top_{Data} \wedge \top_{Data} & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell' \end{cases},\ \textbf{abs}(r \wedge_R \eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'. \begin{cases} \top_{Data} & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell' \end{cases},\ \textbf{abs}(r \wedge_R \eta(\ell))d') \quad \text{(A.4)}
\end{aligned}$$

And $\mathcal{F}[\![\ell : P']\!]$:

$$\begin{aligned}
\mathcal{F}[\![\ell : P']\!] &= \lambda(\eta', d').\ \text{let } (\eta_1, d_1) \leftarrow \mathcal{F}[\![P']\!](\eta', d') \\
&\qquad \text{in } (\eta_1[\ell \mapsto \top_{Data}],\ d_1 \wedge \eta_1(\ell)) \\
&= \lambda(\eta', d').\ \text{let } (\eta_1, d_1) \leftarrow (\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d',\ \textbf{abs}(r)d'))(\eta', d') \\
&\qquad \text{in } (\eta_1[\ell \mapsto \top_{Data}],\ d_1 \wedge \eta_1(\ell)) \\
&= \lambda(\eta', d').\ \text{let } (\eta_1, d_1) \leftarrow (\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d',\ \textbf{abs}(r)d') \\
&\qquad \text{in } (\eta_1[\ell \mapsto \top_{Data}],\ d_1 \wedge \eta_1(\ell)) \\
&= \lambda(\eta', d').((\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d')[\ell \mapsto \top_{Data}], \\
&\qquad \textbf{abs}(r)d' \wedge (\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d')(\ell)) \\
&= \lambda(\eta', d').((\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d')[\ell \mapsto \top_{Data}],\ \textbf{abs}(r)d' \wedge \eta'(\ell) \wedge \textbf{abs}(\eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'. \begin{cases} \top_{Data} & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell' \end{cases},\ \textbf{abs}(r)d' \wedge \top_{Data} \wedge \textbf{abs}(\eta(\ell))d') \\
&= \lambda(\eta', d').(\lambda\ell'. \begin{cases} \top_{Data} & \text{if } \ell = \ell' \\ \eta'(\ell') \wedge \textbf{abs}(\eta(\ell'))d' & \text{if } \ell \neq \ell' \end{cases},\ \textbf{abs}(r \wedge_R \eta(\ell))d') \\
&= (A.4)
\end{aligned}$$

- Case 5. $P = P_1; P_2$

Let $(\eta_1, r_1) = \mathcal{R}[\![P_1]\!]$ and $(\eta_2, r_2) = \mathcal{R}[\![P_2]\!]$. By the induction hypothesis we have

$$\begin{aligned}
\mathcal{F}[\![P_1]\!] &= \textbf{abs}_E(\mathcal{R}[\![P_1]\!]) \\
&= \textbf{abs}_E((\eta_1, r_1)) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta_1(\ell'))d',\ \textbf{abs}(r_1)d')
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}[\![P_2]\!] &= \textbf{abs}_E(\mathcal{R}[\![P_2]\!]) \\
&= \textbf{abs}_E((\eta_2, r_2)) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta_2(\ell'))d',\ \textbf{abs}(r_2)d')
\end{aligned}$$

Now we work on $\textbf{abs}_E(\mathcal{R}[\![P_1; P_2]\!])$.

$$\begin{aligned}
&\textbf{abs}_E(\mathcal{R}[\![P_1; P_2]\!]) = \textbf{abs}_E((\eta_1 \wedge_R (r_1;_R \eta_2), r_1;_R r_2)) \\
&\quad = \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}((\eta_1 \wedge_R (r_1;_R \eta_2))(\ell'))d',\ \textbf{abs}(r_1;_R r_2)d')
\end{aligned} \tag{A.5}$$

And $\mathcal{F}[\![P_1; P_2]\!]$:

$$\begin{aligned}
&\mathcal{F}[\![P_1; P_2]\!] = \lambda(\eta', d').(\mathcal{F}[\![P_1]\!]; \mathcal{F}[\![P_2]\!])(\eta', d') \\
&= \lambda(\eta', d').\mathcal{F}[\![P_2]\!](\mathcal{F}[\![P_1]\!](\eta', d')) \\
&= \lambda(\eta', d').\mathcal{F}[\![P_2]\!]((\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta_1(\ell'))d',\ \textbf{abs}(r_1)d'))(\eta', d')) \\
&= \lambda(\eta', d').\mathcal{F}[\![P_2]\!](\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta_1(\ell'))d',\ \textbf{abs}(r_1)d') \\
&= \lambda(\eta', d').(\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta_2(\ell'))d',\ \textbf{abs}(r_2)d')) \\
&\qquad\qquad (\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta_1(\ell'))d',\ \textbf{abs}(r_1)d') \\
&= \lambda(\eta', d').(\lambda\ell'.(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta_1(\ell'))(d'))(\ell') \wedge \textbf{abs}(\eta_2(\ell'))(\textbf{abs}(r_1)d'), \\
&\qquad\qquad \textbf{abs}(r_2)(\textbf{abs}(r_1)d')) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta_1(\ell'))(d') \wedge \textbf{abs}(\eta_2(\ell'))(\textbf{abs}(r_1)d'),\ \textbf{abs}(r_1;_R r_2)d') \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}((\eta_1 \wedge_R (r_1;_R \eta_2))(\ell'))d',\ \textbf{abs}(r_1;_R r_2)d') \\
&= (A.5)
\end{aligned}$$

- Case 6. $P = \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$
  Let $(\eta_1, r_1) = \mathcal{R}[\![P_1]\!]$ and $(\eta_2, r_2) = \mathcal{R}[\![P_2]\!]$. By the induction hypothesis we have

$$\begin{aligned}
\mathcal{F}[\![P_1]\!] &= \textbf{abs}_E(\mathcal{R}[\![P_1]\!]) \\
&= \textbf{abs}_E((\eta_1, r_1)) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \textbf{abs}(\eta_1(\ell'))d',\ \textbf{abs}(r_1)d')
\end{aligned}$$

$$
\begin{aligned}
\mathcal{F}[\![P_2]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_2]\!]) \\
&= \mathbf{abs}_E((\eta_2, r_2)) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))d',\ \mathbf{abs}(r_2)d')
\end{aligned}
$$

Now we work on $\mathbf{abs}_E(\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!])$.

$$
\begin{aligned}
&\mathbf{abs}_E(\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]) = \mathbf{abs}_E(\mathit{exp}_R(e);_R((\eta_1, r_1) \wedge_R (\eta_2, r_2))) \\
&\quad = \mathbf{abs}_E(\mathit{exp}_R(e);_R((\eta_1, r_1) \wedge_R (\eta_2, r_2))) \\
&\quad = \mathbf{abs}_E((\mathit{exp}_R(e);_R(\eta_1 \wedge_R \eta_2), \mathit{exp}_R(e);_R(r_1 \wedge_R r_2))) \\
&\quad = \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}((\mathit{exp}_R(e);_R(\eta_1 \wedge_R \eta_2))(\ell'))d', \\
&\qquad\qquad \mathbf{abs}(\mathit{exp}_R(e);_R(r_1 \wedge_R r_2))d') \\
&\quad = \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\mathit{exp}_R(e);_R(\eta_1(\ell') \wedge_R \eta_2(\ell')))d', \\
&\qquad\qquad \mathbf{abs}(\mathit{exp}_R(e);_R(r_1 \wedge_R r_2))d')
\end{aligned} \tag{A.6}
$$

And $\mathcal{F}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]$:

$$
\begin{aligned}
&\mathcal{F}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta', d').\ \mathrm{let}\ (\eta_1', d_1') \leftarrow \mathcal{F}[\![P_1]\!](\eta', \mathit{exp}(e)d') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\eta_2', d_2') \leftarrow \mathcal{F}[\![P_2]\!](\eta', \mathit{exp}(e)d') \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{in}\ (\eta_1', d_1') \wedge (\eta_2', d_2') \\
&= \lambda(\eta', d').\ \mathrm{let}\ (\eta_1', d_1') \leftarrow (\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d',\ \mathbf{abs}(r_1)d'))(\eta', \mathit{exp}(e)d') \\
&\qquad\qquad\qquad (\eta_2', d_2') \leftarrow (\lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))d',\ \mathbf{abs}(r_2)d'))(\eta', \mathit{exp}(e)d') \\
&\qquad\qquad \mathrm{in}\ (\eta_1', d_1') \wedge (\eta_2', d_2') \\
&= \lambda(\eta', d').\ \mathrm{let}\ (\eta_1', d_1') \leftarrow (\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))(\mathit{exp}(e)d'),\ \mathbf{abs}(r_1)(\mathit{exp}(e)d')) \\
&\qquad\qquad\qquad (\eta_2', d_2') \leftarrow (\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))(\mathit{exp}(e)d'),\ \mathbf{abs}(r_2)(\mathit{exp}(e)d')) \\
&\qquad\qquad \mathrm{in}\ (\eta_1', d_1') \wedge (\eta_2', d_2') \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))(\mathit{exp}(e)d') \wedge \eta'(\ell') \wedge \mathbf{abs}(\eta_2(\ell'))(\mathit{exp}(e)d'), \\
&\qquad\qquad \mathbf{abs}(r_1)(\mathit{exp}(e)d') \wedge \mathbf{abs}(r_2)(\mathit{exp}(e)d')) \\
&= \lambda(\eta', d').(\lambda\ell'.\eta'(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))(\mathit{exp}(e)d') \wedge \mathbf{abs}(\eta_2(\ell'))(\mathit{exp}(e)d'), \\
&\qquad\qquad \mathbf{abs}(\mathit{exp}_R(e);_R(r_1 \wedge_R r_2))d') \\
&= (A.6)
\end{aligned}
$$

$\square$

Proof of Theorem 4.1.7. The proof is by induction on the structure of P.

- Case 1. $P = \mathsf{skip}$

For this case, we have $(\eta, r) = (\top_{Env_R}, id_R) = \mathcal{R}[\![\mathsf{skip}]\!]$.

$$\begin{aligned}\mathbf{abs}_E(\mathcal{F}^R[\![\mathsf{skip}]\!](\eta', r')) &= \mathbf{abs}_E(id^R(\eta', r')) \\ &= \mathbf{abs}_E((\eta', r')) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(r')d'') \qquad (1)\end{aligned}$$

And

$$\begin{aligned}&\mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_{Env_R}(\ell')), r' ;_R id_R)) \\ &= \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_R), r' ;_R id_R)) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_R))(\ell'))d'', \mathbf{abs}(r' ;_R id_R)d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \top_R))d'', \mathbf{abs}(r')d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' \wedge \top_{Data}, \mathbf{abs}(r')d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(r')d'') \\ &= (1)\end{aligned}$$

- Case 2. $P = x = e$

  For this case, we have $(\eta, r) = (\top_{Env_R}, asgn_R(x, e)) = \mathcal{R}[\![x = e]\!]$.

$$\begin{aligned}&\mathbf{abs}_E(\mathcal{F}^R[\![x = e]\!](\eta', r')) = \mathbf{abs}_E((\eta', asgn^R(x, e)(r'))) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(asgn^R(x, e)(r'))d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(r' ;_R asgn_R(x, e))d'') \qquad (2)\end{aligned}$$

And

$$\begin{aligned}&\mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_{Env_R}(\ell')), r' ;_R asgn_R(x, e))) \\ &= \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_R), r' ;_R asgn_R(x, e))) \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_R))(\ell'))d'', \\ &\qquad \mathbf{abs}(r' ;_R asgn_R(x, e))d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \top_R))d'', \mathbf{abs}(r' ;_R asgn_R(x, e))d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' \wedge \top_{Data}, \mathbf{abs}(r' ;_R asgn_R(x, e))d'') \\ &= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'', \mathbf{abs}(r' ;_R asgn_R(x, e))d'') \\ &= (2)\end{aligned}$$

- Case 3. $P = \mathsf{break}\ \ell$

For this case, we have $(\eta, r) = (\top_{Env_R}[\ell \mapsto id_R], \top_R) = \mathcal{R}[\![\mathsf{break}\ \ell]\!]$.

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{F}^R[\![\mathsf{break}\ \ell]\!](\eta', r')) = \mathbf{abs}_E((\eta'[\ell \mapsto r' \wedge_R \eta'(\ell)], \top_R)) \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'[\ell \mapsto r' \wedge_R \eta'(\ell)](\ell'))d'', \mathbf{abs}(\top_R)d'') \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'[\ell \mapsto r' \wedge_R \eta'(\ell)](\ell'))d'', \top_{Data}) \\
&\quad = \quad \lambda(\eta'', d'').(\Big(\lambda\ell'.\begin{cases} \eta''(\ell) \wedge \mathbf{abs}(r' \wedge_R \eta'(\ell))d'' & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' & \text{if } \ell \neq \ell' \end{cases}\Big), \top_{Data}) \qquad (3)
\end{aligned}$$

And

$$\begin{aligned}
&\mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_{Env_R}[\ell \mapsto id_R](\ell')), r' ;_R \top_R))) \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \top_{Env_R}[\ell \mapsto id_R](\ell')))(\ell'))d'', \\
&\qquad\qquad\qquad \mathbf{abs}(r' ;_R \top_R)d'') \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \top_{Env_R}[\ell \mapsto id_R](\ell')))d'', \top_{Data}) \\
&\quad = \quad \lambda(\eta'', d'').(\Big(\lambda\ell'.\begin{cases} \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell) \wedge_R (r' ;_R id_R))d'' & \text{if } \ell = \ell' \\ \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell) \wedge_R (r' ;_R \top_R))d'' & \text{if } \ell \neq \ell' \end{cases}\Big), \top_{Data}) \\
&\quad = \quad \lambda(\eta'', d'').(\Big(\lambda\ell'.\begin{cases} \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell) \wedge_R r')d'' & \text{if } \ell = \ell' \\ \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell'))d'' & \text{if } \ell \neq \ell' \end{cases}\Big), \top_{Data}) \\
&\quad = \quad (3)
\end{aligned}$$

- Case 4. $P = \ell : P'$
  Let $(\eta, r) = \mathcal{R}[\![P']\!]$. By the induction hypothesis, we obtain

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{F}^R[\![P']\!](\eta', r')) \\
&\quad = \quad \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')), r' ;_R r)) \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))(\ell'))d'', \mathbf{abs}(r' ;_R r)d'') \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))d'', \mathbf{abs}(r' ;_R r)d'') \qquad (4)
\end{aligned}$$

Let $(\eta_1, r_1) = \mathcal{F}^R[\![P']\!](\eta', r')$. Then we get

$$\begin{aligned}
\mathbf{abs}_E(\mathcal{F}^R[\![P']\!](\eta', r')) &= \mathbf{abs}_E((\eta_1, r_1)) \\
&= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_1(\ell'))d'', \mathbf{abs}(r_1)d'') \qquad (5)
\end{aligned}$$

Since $(4) = (5)$, we obtain

$$\mathbf{abs}(r' ;_R r)d'' \quad = \quad \mathbf{abs}(r_1)d''$$

and

$$\eta''(\ell') \wedge \textbf{abs}(\eta'(\ell') \wedge_R (r' \,;_R\, \eta(\ell')))d'' \quad = \quad \eta''(\ell') \wedge \textbf{abs}(\eta_1(\ell'))d''$$

When $\ell' = \ell$, using the legality condition, we get

$$\begin{array}{rrcl} & \eta''(\ell) \wedge \textbf{abs}(\eta'(\ell) \wedge_R (r' \,;_R\, \eta(\ell)))d'' & = & \eta''(\ell) \wedge \textbf{abs}(\eta_1(\ell))d'' \\ \Rightarrow & \top_{Data} \wedge \textbf{abs}(\top_R \wedge_R (r' \,;_R\, \eta(\ell)))d'' & = & \top_{Data} \wedge \textbf{abs}(\eta_1(\ell))d'' \\ \Rightarrow & \textbf{abs}(r' \,;_R\, \eta(\ell))d'' & = & \textbf{abs}(\eta_1(\ell))d'' \end{array}$$

Now we work on $\ell : P'$:

$$\begin{array}{rl} \multicolumn{2}{l}{\textbf{abs}_E(\mathcal{F}^R[\![\ell : P']\!](\eta', r'))} \\ = & \textbf{abs}_E((\eta_1[\ell \mapsto \top_R], r_1 \wedge_R \eta_1(\ell))) \\ = & \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \textbf{abs}(\eta_1[\ell \mapsto \top_R](\ell'))d'', \textbf{abs}(r_1 \wedge_R \eta_1(\ell))d'') \\ = & \lambda(\eta'', d'').(\lambda\ell'. \begin{cases} \eta''(\ell) \wedge \textbf{abs}(\top_R)d'' & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \textbf{abs}(\eta_1(\ell'))d'' & \text{if } \ell \neq \ell' \end{cases}, \\ & \qquad \textbf{abs}(r_1)d'' \wedge \textbf{abs}(\eta_1(\ell))d'') \\ = & \lambda(\eta'', d'').(\lambda\ell'. \begin{cases} \eta''(\ell) & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \textbf{abs}(\eta'(\ell') \wedge_R (r' \,;_R\, \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, \\ & \qquad \textbf{abs}(r' \,;_R\, r)d'' \wedge \textbf{abs}(r' \,;_R\, \eta(\ell))d'') \\ = & \lambda(\eta'', d'').(\lambda\ell'. \begin{cases} \eta''(\ell) & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \textbf{abs}(\eta'(\ell') \wedge_R (r' \,;_R\, \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, \\ & \qquad \textbf{abs}(r' \,;_R\, (r \wedge_R \eta(\ell)))d'') \qquad (6) \end{array}$$

And using the fact that $\mathcal{R}[\![\ell : P']\!] = (\eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))$, we have

$$\begin{aligned}
&\mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta[\ell \mapsto \top_R](\ell')), r' ;_R (r \wedge_R \eta(\ell)))) \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta[\ell \mapsto \top_R](\ell')))(\ell'))d'', \\
&\qquad\qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta[\ell \mapsto \top_R](\ell')))d'', \\
&\qquad\qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\begin{cases} \eta''(\ell) \wedge \mathbf{abs}(\eta'(\ell) \wedge_R (r' ;_R \top_R))d'' & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, \\
&\qquad\qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\begin{cases} \eta''(\ell) \wedge \mathbf{abs}(\top_R \wedge_R (r' ;_R \top_R))d'' & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, \\
&\qquad\qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\begin{cases} \eta''(\ell) & \text{if } \ell = \ell' \\ \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta(\ell')))d'' & \text{if } \ell \neq \ell' \end{cases}, \\
&\qquad\qquad \mathbf{abs}(r' ;_R (r \wedge_R \eta(\ell)))d'') \\
&\quad = (6)
\end{aligned}$$

- Case 5. $P = P_1; P_2$

  Let $(\eta_1, r_1) = \mathcal{R}[\![P_1]\!]$, $(\eta_2, r_2) = \mathcal{R}[\![P_2]\!]$, $(\eta_a, r_a) = \mathcal{F}^R[\![P_1]\!](\eta', r')$, and $(\eta_b, r_b) = \mathcal{F}^R[\![P_2]\!](\eta_a, r_a)$. By the induction hypothesis, we have

$$\begin{aligned}
\mathbf{abs}_E(\mathcal{F}^R[\![P_1]\!](\eta', r')) &= \mathbf{abs}_E((\eta_a, r_a)) \\
&= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'',\ \mathbf{abs}(r_a)d'')
\end{aligned}$$

  and

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{F}^R[\![P_1]\!](\eta', r')) = \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta_1(\ell')),\ r' ;_R r_1)) \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' ;_R \eta_1(\ell')))(\ell'))d'',\ \mathbf{abs}(r' ;_R r_1)d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' ;_R \eta_1(\ell')))d'',\ \mathbf{abs}(r' ;_R r_1)d'')
\end{aligned}$$

  Similarly, for P_2

$$\begin{aligned}
\mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta_a, r_a)) &= \mathbf{abs}_E((\eta_b, r_b)) \\
&= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'',\ \mathbf{abs}(r_b)d'')
\end{aligned}$$

and

$$
\begin{aligned}
&\mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta_a, r_a)) = \mathbf{abs}_E((\lambda\ell'.\eta_a(\ell') \wedge_R (r_a \,;_R \eta_2(\ell')), r_a \,;_R r_2)) \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta_a(\ell') \wedge_R (r_a \,;_R \eta_2(\ell')))(\ell'))d'',\ \mathbf{abs}(r_a \,;_R r_2)d'') \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell') \wedge_R (r_a \,;_R \eta_2(\ell')))d'',\ \mathbf{abs}(r_a \,;_R r_2)d'')
\end{aligned}
$$

These give us the equalities

$$
\begin{aligned}
\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'' &= \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' \,;_R \eta_1(\ell')))d'' \\
\mathbf{abs}(r_a)d'' &= \mathbf{abs}(r' \,;_R r_1)d'' \\
\eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'' &= \eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell') \wedge_R (r_a \,;_R \eta_2(\ell')))d'' \\
\mathbf{abs}(r_b)d'' &= \mathbf{abs}(r_a \,;_R r_2)d''
\end{aligned}
$$

Now, returning to $P_1; P_2$, we have

$$
\begin{aligned}
&\mathbf{abs}_E(\mathcal{F}^R[\![P_1; P_2]\!](\eta', r')) = \mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\mathcal{F}^R[\![P_1]\!](\eta', r'))) \\
&\quad = \quad \mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta_a, r_a)) \\
&\quad = \quad \mathbf{abs}_E((\eta_b, r_b)) \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'',\ \mathbf{abs}(r_b)d'') \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell') \wedge_R (r_a \,;_R \eta_2(\ell')))d'',\ \mathbf{abs}(r_a \,;_R r_2)d'') \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'' \wedge \mathbf{abs}(\eta_2(\ell'))(\mathbf{abs}(r_a)d''),\ \mathbf{abs}(r_2)(\mathbf{abs}(r_a)d'')) \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' \,;_R \eta_1(\ell')))d'' \\
&\qquad\qquad\qquad \wedge \mathbf{abs}(\eta_2(\ell'))(\mathbf{abs}(r' \,;_R r_1)d''), \\
&\qquad\qquad\quad \mathbf{abs}(r_2)(\mathbf{abs}(r' \,;_R r_1)d'')) \\
&\quad = \quad \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell')) \wedge (\mathbf{abs}(r'); \mathbf{abs}(\eta_1(\ell')))d'' \\
&\qquad\qquad\qquad \wedge(\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(\eta_2(\ell')))d'', \\
&\qquad\qquad\quad (\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(r_2))d'') \qquad (7)
\end{aligned}
$$

for the left-hand-side of the equivalence. And using the fact that $\mathcal{R}[\![P_1; P_2]\!] = (\eta_1 \wedge_R (r_1 \,;_R \eta_2),\ r_1 ;_R r_2)$, for the right-hand-side of the equivalence we have

$$
\begin{aligned}
&\mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' \,;_R (\eta_1 \wedge_R (r_1 \,;_R \eta_2))(\ell')), r' \,;_R (r_1;_R r_2)))\\
&\quad= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (r' \,;_R (\eta_1 \wedge_R (r_1 \,;_R \eta_2))(\ell')))(\ell'))d'',\\
&\qquad\qquad \mathbf{abs}(r' \,;_R (r_1;_R r_2))d'')\\
&\quad= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' \,;_R (\eta_1 \wedge_R (r_1 \,;_R \eta_2))(\ell')))d'',\\
&\qquad\qquad \mathbf{abs}(r' \,;_R (r_1;_R r_2))d'')\\
&\quad= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell')) \wedge (\mathbf{abs}(r'); \mathbf{abs}((\eta_1 \wedge_R (r_1 \,;_R \eta_2))(\ell')))d'',\\
&\qquad\qquad (\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(r_2))d'')\\
&\quad= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell')) \wedge (\mathbf{abs}(r'); \mathbf{abs}(\eta_1(\ell')))d''\\
&\qquad\qquad\quad \wedge(\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(\eta_2(\ell')))d'',\\
&\qquad\qquad (\mathbf{abs}(r'); \mathbf{abs}(r_1); \mathbf{abs}(r_2))d'')\\
&\quad= (7)
\end{aligned}
$$

- Case 6. $P = \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$
  Let $(\eta_1, r_1) = \mathcal{R}[\![P_1]\!]$, $(\eta_2, r_2) = \mathcal{R}[\![P_2]\!]$, $(\eta_a, r_a) = \mathcal{F}^R[\![P_1]\!](\eta', exp^R(e)r')$, and $(\eta_b, r_b) = \mathcal{F}^R[\![P_2]\!](\eta', exp^R(e)r')$. By the induction hypothesis, we have

$$
\begin{aligned}
\mathbf{abs}_E(\mathcal{F}^R[\![P_1]\!](\eta', exp^R(e)r')) &= \mathbf{abs}_E((\eta_a, r_a))\\
&= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'',\ \mathbf{abs}(r_a)d'')
\end{aligned}
$$

  and

$$
\begin{aligned}
&\mathbf{abs}_E(\mathcal{F}^R[\![P_1]\!](\eta', exp^R(e)r'))\\
&\quad= \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (exp^R(e)r' \,;_R \eta_1(\ell')),\ exp^R(e)r' \,;_R r_1))\\
&\quad= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (exp^R(e)r' \,;_R \eta_1(\ell')))(\ell'))d'',\\
&\qquad\qquad \mathbf{abs}(exp^R(e)r' \,;_R r_1)d'')\\
&\quad= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_1(\ell')))d'',\\
&\qquad\qquad \mathbf{abs}(r';_R exp_R(e);_R r_1)d'')
\end{aligned}
$$

  Similarly, for P_2

$$
\begin{aligned}
\mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta', exp^R(e)r')) &= \mathbf{abs}_E((\eta_b, r_b))\\
&= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'',\ \mathbf{abs}(r_b)d'')
\end{aligned}
$$

and

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{F}^R[\![P_2]\!](\eta', exp^R(e)r')) \\
&\quad = \mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (exp^R(e)r' \;;_R \eta_2(\ell')),\ exp^R(e)r' \;;_R r_2)) \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}((\lambda\ell'.\eta'(\ell') \wedge_R (exp^R(e)r' \;;_R \eta_2(\ell')))(\ell'))d'', \\
&\qquad\qquad \mathbf{abs}(exp^R(e)r' \;;_R r_2)d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_2(\ell')))d'', \\
&\qquad\qquad \mathbf{abs}(r';_R exp_R(e);_R r_2)d'')
\end{aligned}$$

These give us the equalities

$$\begin{aligned}
\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell'))d'' &= \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_1(\ell')))d'' \\
\mathbf{abs}(r_a)d'' &= \mathbf{abs}(r';_R exp_R(e);_R r_1)d'' \\
\eta''(\ell') \wedge \mathbf{abs}(\eta_b(\ell'))d'' &= \eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_2(\ell')))d'' \\
\mathbf{abs}(r_b)d'' &= \mathbf{abs}(r';_R exp_R(e);_R r_2)d''
\end{aligned}$$

Now, returning to $\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$, we have

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{F}^R[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!](\eta', r')) = \mathbf{abs}_E((\eta_a \wedge_R \eta_b, r_a \wedge_R r_b)) \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta_a(\ell') \wedge_R \eta_b(\ell'))d'', \mathbf{abs}(r_a \wedge_R r_b)d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_1(\ell')))d'' \\
&\qquad\qquad \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r';_R exp_R(e);_R \eta_2(\ell')))d'', \\
&\qquad\qquad \mathbf{abs}(r';_R exp_R(e);_R r_1)d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R r_2)d'') \\
&\quad = \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R \eta_1(\ell'))d'' \\
&\qquad\qquad \wedge \mathbf{abs}(r';_R exp_R(e);_R \eta_2(\ell'))d'', \\
&\qquad\qquad \mathbf{abs}(r';_R exp_R(e);_R r_1)d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R r_2)d'')
\end{aligned} \tag{8}$$

And using the fact that

$$\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = (exp_R(e);_R (\eta_1 \wedge_R \eta_2), exp_R(e);_R (r_1 \wedge_R r_2))$$

we get

$$\begin{aligned}
&\mathbf{abs}_E((\lambda\ell'.\eta'(\ell') \wedge_R (r' \,;_R (exp_R(e);_R (\eta_1(\ell') \wedge_R \eta_2(\ell')))), r' \,;_R (exp_R(e);_R (r_1 \wedge_R r_2)))) \\
&= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell') \wedge_R (r' \,;_R (exp_R(e);_R (\eta_1(\ell') \wedge_R \eta_2(\ell')))))d'', \\
&\qquad \mathbf{abs}(r' \,;_R (exp_R(e);_R (r_1 \wedge_R r_2)))d'') \\
&= \lambda(\eta'', d'').(\lambda\ell'.\eta''(\ell') \wedge \mathbf{abs}(\eta'(\ell'))d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R \eta_1(\ell'))d'' \\
&\qquad \wedge \mathbf{abs}(r';_R exp_R(e);_R \eta_2(\ell'))d'', \\
&\qquad \mathbf{abs}(r';_R exp_R(e);_R r_1)d'' \wedge \mathbf{abs}(r';_R exp_R(e);_R r_2)d'') \\
&= (8)
\end{aligned}$$

□

Proof of Theorem 4.2.1. To show that a representation is exact (i.e. **abs** is an isomorphism between R and *DFFun*), we need to prove two claims:

1. R is adequate (i.e. **abs** defines a homomorphism)
2. **abs** is injective. (i.e. $\forall r_1, r_2 \in R, r_1 \neq r_2 \Rightarrow \mathbf{abs}(r_1) \neq \mathbf{abs}(r_2)$)

Claim 1: R for RD is adequate. Proof:

- $\mathbf{abs}(\top_R) = \lambda D.\top_{Data}$ holds by definition.
- $\mathbf{abs}(id_R) = \mathbf{abs}((\varnothing, \varnothing)) = \lambda D.\varnothing \cup (D \setminus \varnothing) = \lambda D.D = id$
- $\mathbf{abs}(asgn_R(n, x, e)) = \mathbf{abs}((\{x\}, \{n\})) = \lambda D.\{n\} \cup (D \setminus \{x\}) = \lambda D.\{n\} \cup (D \setminus D_x) = asgn(n, x, e)$
- $\mathbf{abs}(exp_R(e)) = \mathbf{abs}((\varnothing, \varnothing)) = \lambda D.\varnothing \cup (D \setminus \varnothing) = \lambda D.D = exp(e)$
- $$\begin{aligned}\mathbf{abs}((K_1, G_1);_R (K_2, G_2)) &= \mathbf{abs}((K_1 \cup K_2, G_2 \cup (G_1 \setminus K_2))) \\ &= \lambda D.G_2 \cup (G_1 \setminus K_2) \cup (D \setminus (K_1 \cup K_2)) \\ &= \lambda D.G_2 \cup (G_1 \setminus K_2) \cup ((D \setminus K_1) \cap (D \setminus K_2)) \qquad (1)\end{aligned}$$

  $$\begin{aligned}\mathbf{abs}((K_1, G_1)); \mathbf{abs}((K_2, G_2)) &= (\lambda D.G_1 \cup (D \setminus K_1)); (\lambda D.G_2 \cup (D \setminus K_2)) \\ &= \lambda D.G_2 \cup ((G_1 \cup (D \setminus K_1)) \setminus K_2) \\ &= \lambda D.G_2 \cup (G_1 \setminus K_2) \cup ((D \setminus K_1) \setminus K_2) \\ &= \lambda D.G_2 \cup (G_1 \setminus K_2) \cup ((D \setminus K_1) \cap (D \setminus K_2)) \\ &= (1)\end{aligned}$$

- $$\begin{aligned}\mathbf{abs}((K_1, G_1) \wedge_R (K_2, G_2)) &= \mathbf{abs}((K_1 \cap K_2, G_1 \cup G_2)) \\ &= \lambda D.G_1 \cup G_2 \cup (D \setminus (K_1 \cap K_2)) \qquad (2)\end{aligned}$$

$$
\begin{aligned}
\textbf{abs}((K_1, G_1)) \wedge \textbf{abs}((K_2, G_2)) &= (\lambda D.G_1 \cup (D \setminus K_1)) \wedge (\lambda D.G_2 \cup (D \setminus K_2)) \\
&= \lambda D.G_1 \cup (D \setminus K_1) \cup G_2 \cup (D \setminus K_2) \\
&= \lambda D.G_1 \cup G_2 \cup (D \setminus (K_1 \cap K_2)) \\
&= (2)
\end{aligned}
$$

Therefore, R for RD is adequate.

Claim 2: **abs** is injective.Proof:

By contradiction. Let $r_1 = (K_1, G_1)$, $r_2 = (K_2, G_2)$ and $r_1 \neq r_2$, which implies $K_1 \neq K_2$ and/or $G_1 \neq G_2$. Assume $\textbf{abs}(r_1) = \textbf{abs}(r_2)$. Then we have

$$\lambda D.G_1 \cup (D \setminus K_1) = \lambda D.G_2 \cup (D \setminus K_2)$$

which means, for all $D \in Data$,

$$G_1 \cup (D \setminus K_1) = G_2 \cup (D \setminus K_2)$$

Now there are two cases to consider: $K_1 = K_2$ and $K_1 \neq K_2$.

- For the first case, take D to be the empty set. Then we get $G_1 = G_2$. But this conflicts with our initial assumption.
- For the second case, without loss of generality, assume $K_1 \setminus K_2 \neq \varnothing$. We can pick D to be $\{n\}$ for some $n \in Node$ such that $n : x = e$, $x \in (K_1 \setminus K_2)$ and $n \notin G_1$. Then we end up with the equality
$$G_1 \cup \varnothing = G_2 \cup \{n\}$$
which is impossible because G_1 does not include n. □

Proof of Theorem 4.2.5. We provide the sketch of the proof here. We first show that R is adequate.

- $\textbf{abs}(\top_R) = \lambda M.\top_{Data}$ holds by definition.
- $$\begin{aligned}\textbf{abs}(id_R) = \textbf{abs}(\lambda v.\varnothing_{sur}) &= \lambda S.\lambda v.\,\text{let } C_{must} \leftarrow \textit{semicolon}(S, S(v)_{must}, \varnothing_{sur}) \\ &\qquad \text{in if } C = \{i\} \text{ then } i \text{ else } \bot \\ &= \lambda S.\lambda v.\,\text{let } C_{must} \leftarrow S(v)_{must} \\ &\qquad \text{in if } C = \{i\} \text{ then } i \text{ else } \bot \\ &= \lambda S.\lambda v.S(v) \\ &= id\end{aligned}$$
- $$\begin{aligned}&\textbf{abs}(asgn_R(n, x, e)) = \textbf{abs}(\lambda v.\varnothing_{sur}[x \mapsto \{e\}_{must}]) \\ &\quad = \lambda S.\lambda v.\,\text{let } C_{must} \leftarrow \textit{semicolon}(S, S(v)_{must}, (\lambda v.\varnothing_{sur}[x \mapsto \{e\}_{must}])(v)) \\ &\qquad\qquad \text{in if } C = \{i\} \text{ then } i \text{ else } \bot\end{aligned}$$

$$= \lambda S.\lambda v. \begin{cases} S(v) & \text{if } v \neq x \\ \mathit{update}(S, \{e\}) & \text{if } v = x \end{cases}$$
$$= \lambda S.S[x \mapsto \text{if } \mathit{isConstant}(e, S) \text{ then } \mathit{consVal}(e, M) \text{ else } \bot]$$
$$= \mathit{asgn}(n, x, e)$$

- $\mathbf{abs}(\mathit{exp}_R(e)) = \mathbf{abs}(\lambda v.\varnothing_{sur}) = \lambda S.S = \mathit{exp}(e)$

- $$\begin{aligned} \mathbf{abs}(M_1 \wedge_R M_2) &= \mathbf{abs}(\lambda v.M_1(v) \wedge_R M_2(v)) \\ &= \lambda S.\lambda v.\text{ let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, M_1(v) \wedge_R M_2(v)) \\ &\qquad\quad \text{in if } C = \{i\} \text{ then } i \text{ else } \bot \\ &= (1) \end{aligned}$$

  and

  $$\begin{aligned} \mathbf{abs}(M_1) \wedge \mathbf{abs}(M_2) &= \begin{pmatrix} \lambda S.\lambda v.\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, M_1(v)) \\ \text{in if } C = \{i\} \text{ then } i \text{ else } \bot \end{pmatrix} \wedge \\ &\quad \begin{pmatrix} \lambda S.\lambda v.\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, M_2(v)) \\ \text{in if } C = \{i\} \text{ then } i \text{ else } \bot \end{pmatrix} \\ &= (2) \end{aligned}$$

  Showing that $(1) = (2)$ is a straightforward case analysis based on the annotations of the values obtained from $M_1(v)$ and $M_2(v)$.

- $$\begin{aligned} \mathbf{abs}(M_1;_R M_2) &= \mathbf{abs}(\lambda v.\mathit{semicolon}(M_1, M_1(v), M_2(v))) \\ &= \lambda S.\lambda v.\text{ let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, \mathit{semicolon}(M_1, M_1(v), M_2(v))) \\ &\qquad\quad \text{in if } C = \{i\} \text{ then } i \text{ else } \bot \\ &= (3) \end{aligned}$$

  and

  $$\begin{aligned} \mathbf{abs}(M_1); \mathbf{abs}(M_2) &= \begin{pmatrix} \lambda S.\lambda v.\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, M_1(v)) \\ \text{in if } C = \{i\} \text{ then } i \text{ else } \bot \end{pmatrix}; \\ &\quad \begin{pmatrix} \lambda S.\lambda v.\text{let } C_{must} \leftarrow \mathit{semicolon}(S, S(v)_{must}, M_2(v)) \\ \text{in if } C = \{i\} \text{ then } i \text{ else } \bot \end{pmatrix} \\ &= (4) \end{aligned}$$

  Showing that $(3) = (4)$ is a straightforward case analysis based on the annotations of the values obtained from $M_1(v)$ and $M_2(v)$.

Next step of the proof requires showing that the representations uniquely represent functions. This part in essence follows the same principles of the corresponding proof of RD (Section 4.2.1). □

Proof of Theorem 4.3.1. The proof is by induction on the structure of P.

- Case 1. $P = \mathsf{skip}$

$$\begin{aligned}
\mathbf{abs}_E(\mathcal{R}[\![\mathsf{skip}]\!]) = \mathbf{abs}_E((\top_{Env_R},\ id_R))& \\
= \quad & \lambda(\eta', d').(\eta', \mathbf{abs}(id_R)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\top_{Env_R}(\ell'))(\eta'(\ell'))) \\
= \quad & \lambda(\eta', d').(\eta',\ d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \top_{Data}) \\
= \quad & \lambda(\eta', d').(\eta',\ d') \\
= \quad & \mathcal{B}[\![\mathsf{skip}]\!]
\end{aligned}$$

- Case 2. $P = x = e$

$$\begin{aligned}
\mathbf{abs}_E(\mathcal{R}[\![x = e]\!]) = \mathbf{abs}_E((\top_{Env_R},\ asgn_R(x, e)))& \\
= \quad & \lambda(\eta', d').(\eta', \mathbf{abs}(asgn_R(x, e))d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\top_{Env_R}(\ell'))(\eta'(\ell'))) \\
= \quad & \lambda(\eta', d').(\eta',\ asgn(x, e)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \top_{Data}) \\
= \quad & \lambda(\eta', d').(\eta',\ asgn(x, e)d') \\
= \quad & \mathcal{B}[\![x = e]\!]
\end{aligned}$$

- Case 3. $P = \mathsf{break}\ \ell$

$$\begin{aligned}
\mathbf{abs}_E(\mathcal{R}[\![\mathsf{break}\ \ell]\!]) = \mathbf{abs}_E((\top_{Env_R}[\ell \mapsto id_R],\ \top_R))& \\
= \quad & \lambda(\eta', d').(\eta', \mathbf{abs}(\top_R)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\top_{Env_R}[\ell \mapsto id_R](\ell'))(\eta'(\ell'))) \\
= \quad & \lambda(\eta', d').(\eta',\ \top_{Data} \wedge \mathbf{abs}(id_R)(\eta'(\ell))) \\
= \quad & \lambda(\eta', d').(\eta',\ \eta'(\ell)) \\
= \quad & \mathcal{B}[\![\mathsf{break}\ \ell]\!]
\end{aligned}$$

- Case 4. $P = \ell : P'$
  Let $(\eta, r) = \mathcal{R}[\![P']\!]$. By the induction hypothesis we have

$$\begin{aligned}
\mathcal{B}[\![P']\!] = \mathbf{abs}_E(\mathcal{R}[\![P']\!]) = \mathbf{abs}_E((\eta, r))& \\
= \quad & \lambda(\eta', d').(\eta', \mathbf{abs}(r)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta(\ell'))(\eta'(\ell')))
\end{aligned}$$

Now we work on $\ell : P'$.

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{R}[\![\ell : P']\!]) = \mathbf{abs}_E((\eta[\ell \mapsto \top_R], r \wedge_R \eta(\ell))) \\
&\quad = \lambda(\eta', d').(\eta', \mathbf{abs}(r \wedge_R \eta(\ell))d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta[\ell \mapsto \top_R](\ell'))(\eta'(\ell'))) \\
&\quad = \lambda(\eta', d').(\eta', \mathbf{abs}(r)d' \wedge \mathbf{abs}(\eta(\ell))d' \wedge \bigwedge_{\ell' \in \text{Label}, \ell' \neq \ell} \mathbf{abs}(\eta(\ell'))(\eta'(\ell'))) \qquad (1)
\end{aligned}$$

And

$$\begin{aligned}
\mathcal{B}[\![\ell : P']\!] &= \lambda(\eta', d').\ \text{let } (\eta_1, d_1) \leftarrow \mathcal{B}[\![P']\!](\eta'[\ell \mapsto d'], d') \\
&\qquad\qquad \text{in } (\eta_1[\ell \mapsto \top_{Data}],\ d_1) \\
&= \lambda(\eta', d').\ \text{let } (\eta_1, d_1) \leftarrow (\eta'[\ell \mapsto d'], \mathbf{abs}(r)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta(\ell'))(\eta'[\ell \mapsto d'](\ell'))) \\
&\qquad \text{in } (\eta_1[\ell \mapsto \top_{Data}],\ d_1) \\
&= \lambda(\eta', d').(\eta'[\ell \mapsto \top_{Data}],\ \mathbf{abs}(r)d' \wedge \mathbf{abs}(\eta(\ell))d' \wedge \bigwedge_{\ell' \in \text{Label}, \ell' \neq \ell} \mathbf{abs}(\eta(\ell'))(\eta'(\ell')))
\end{aligned}$$

Because we require all the programs to be legal, the incoming environment η' has ℓ mapped to $\top_{Data}$. This means that $\eta' = \eta'[\ell \mapsto \top_{Data}]$. So

$$\begin{aligned}
&= \lambda(\eta', d').(\eta',\ \mathbf{abs}(r)d' \wedge \mathbf{abs}(\eta(\ell))d' \wedge \bigwedge_{\ell' \in \text{Label}, \ell' \neq \ell} \mathbf{abs}(\eta(\ell'))(\eta'(\ell'))) \\
&= (1)
\end{aligned}$$

- Case 5. $P = P_1; P_2$

  Let $(\eta_1, r_1) = \mathcal{R}[\![P_1]\!]$ and $(\eta_2, r_2) = \mathcal{R}[\![P_2]\!]$. By the induction hypothesis we have

$$\begin{aligned}
\mathcal{B}[\![P_1]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_1]\!]) \\
&= \mathbf{abs}_E((\eta_1, r_1)) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_1)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell')))
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{B}[\![P_2]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_2]\!]) \\
&= \mathbf{abs}_E((\eta_2, r_2)) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_2)d' \wedge \bigwedge_{\ell' \in \text{Label}} \mathbf{abs}(\eta_2(\ell'))(\eta'(\ell')))
\end{aligned}$$

Now we work on $P_1; P_2$.

$$\begin{aligned}
&\mathbf{abs}_E(\mathcal{R}[\![P_1;P_2]\!]) = \mathbf{abs}_E((\eta_1 \wedge_R (\eta_2 ;_R r_1), r_2 ;_R r_1)) \\
&\quad = \lambda(\eta', d').(\eta', \mathbf{abs}(r_2 ;_R r_1) d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}((\eta_1 \wedge_R (\eta_2 ;_R r_1))(\ell'))(\eta'(\ell'))) \\
&\quad = \lambda(\eta', d').(\eta', \mathbf{abs}(r_2 ;_R r_1) d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} (\mathbf{abs}(\eta_1(\ell'))(\eta'(\ell')) \\
&\qquad\qquad \wedge \mathbf{abs}(\eta_2(\ell') ;_R r_1)(\eta'(\ell')))) \qquad (3)
\end{aligned}$$

And

$$\begin{aligned}
&\mathcal{B}[\![P_1;P_2]\!] = \lambda(\eta', d').(\mathcal{B}[\![P_2]\!];\mathcal{B}[\![P_1]\!])(\eta', d') \\
&\quad = \lambda(\eta', d').\mathcal{B}[\![P_1]\!](\mathcal{B}[\![P_2]\!](\eta', d')) \\
&\quad = \lambda(\eta', d').\mathcal{B}[\![P_1]\!](\eta', \mathbf{abs}(r_2) d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta_2(\ell'))(\eta'(\ell'))) \\
&\quad = \lambda(\eta', d').\Big(\eta', \mathbf{abs}(r_1)\big(\mathbf{abs}(r_2) d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta_2(\ell'))(\eta'(\ell'))\big) \\
&\qquad\qquad \wedge \big(\bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell'))\big)\Big) \\
&\quad = \lambda(\eta', d').\Big(\eta', \mathbf{abs}(r_2 ;_R r_1) d' \wedge \big(\bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta_2(\ell') ;_R r_1)(\eta'(\ell'))\big) \\
&\qquad\qquad \wedge \big(\bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell'))\big)\Big) \\
&\quad = \lambda(\eta', d').\Big(\eta', \mathbf{abs}(r_2 ;_R r_1) d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} (\mathbf{abs}(\eta_2(\ell') ;_R r_1)(\eta'(\ell')) \wedge \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell')))\Big) \\
&\quad = (3)
\end{aligned}$$

We note that we used the distributivity property above.

- Case 6. $P = \mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$
  Let $(\eta_1, r_1) = \mathcal{R}[\![P_1]\!]$ and $(\eta_2, r_2) = \mathcal{R}[\![P_2]\!]$. By the induction hypothesis we have

$$\begin{aligned}
\mathcal{B}[\![P_1]\!] &= \mathbf{abs}_E(\mathcal{R}[\![P_1]\!]) \\
&= \mathbf{abs}_E((\eta_1, r_1)) \\
&= \lambda(\eta', d').(\eta', \mathbf{abs}(r_1) d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \mathbf{abs}(\eta_1(\ell'))(\eta'(\ell')))
\end{aligned}$$

$$\begin{array}{rcl}\mathcal{B}[\![P_2]\!] & = & \textbf{abs}_E(\mathcal{R}[\![P_2]\!]) \\ & = & \textbf{abs}_E((\eta_2, r_2)) \\ & = & \lambda(\eta', d').(\eta', \textbf{abs}(r_2)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \textbf{abs}(\eta_2(\ell'))(\eta'(\ell')))\end{array}$$

Now we work on $\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2$.

$$\begin{array}{l}\textbf{abs}_E(\mathcal{R}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!]) = \textbf{abs}_E(((\eta_1 \wedge_R \eta_2);_R exp_R(e), (r_1 \wedge_R r_2);_R exp(e))) \\ \quad = \quad \lambda(\eta', d').\Big(\eta', \textbf{abs}((r_1 \wedge_R r_2);_R exp(e))d' \\ \qquad\qquad\qquad \wedge \bigwedge_{\ell' \in \mathrm{Label}} \textbf{abs}(((\eta_1 \wedge_R \eta_2);_R exp_R(e))(\ell'))(\eta'(\ell'))\Big) \\ \quad = \quad \lambda(\eta', d').\Big(\eta', exp(e)(\textbf{abs}(r_1 \wedge_R r_2)d') \\ \qquad\qquad\qquad \wedge exp(e)\Big(\bigwedge_{\ell' \in \mathrm{Label}} \textbf{abs}(\eta_1(\ell') \wedge_R \eta_2(\ell'))(\eta'(\ell'))\Big)\Big) \qquad (4)\end{array}$$

Let $(\eta_1', d_1') = \mathcal{B}[\![P_1]\!](\eta', d')$ and $(\eta_2', d_2') = \mathcal{B}[\![P_2]\!](\eta', d')$. Then

$$\begin{array}{l}\mathcal{B}[\![\mathsf{if}(e)\ P_1\ \mathsf{else}\ P_2]\!] = \lambda(\eta', d').(\eta', exp(e)(d_1 \wedge d_2)) \\ \quad = \quad \lambda(\eta', d').\Big(\eta', exp(e)((\textbf{abs}(r_1)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \textbf{abs}(\eta_1(\ell'))(\eta'(\ell'))) \\ \qquad\qquad\qquad \wedge(\textbf{abs}(r_2)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \textbf{abs}(\eta_2(\ell'))(\eta'(\ell'))))\Big) \\ \quad = \quad \lambda(\eta', d').\Big(\eta', exp(e)(\textbf{abs}(r_1 \wedge_R r_2)d' \wedge \bigwedge_{\ell' \in \mathrm{Label}} \textbf{abs}(\eta_1(\ell') \wedge_R \eta_2(\ell'))(\eta'(\ell')))\Big) \\ \quad = \quad \lambda(\eta', d').\Big(\eta', exp(e)(\textbf{abs}(r_1 \wedge_R r_2)d') \wedge exp(e)\Big(\bigwedge_{\ell' \in \mathrm{Label}} \textbf{abs}(\eta_1(\ell') \wedge_R \eta_2(\ell'))(\eta'(\ell'))\Big)\Big) \\ \quad = \quad (4)\end{array}$$

We note that we used the distributivity property above.

□

A.2 Proofs of Theorems in Chapter 5

A.2.1 Record Language

The record calculus λ^{rec}_{poly} satisfies the following standard lemmas.

Lemma A.2.1 (Weakening/Strengthening). *If $\Delta(w) = \Delta'(w)$ for all $w \in FV(e)$, then $\Delta \vdash_R e : T$ iff $\Delta' \vdash_R e : T$.*

Lemma A.2.2 (Substitution). *If $\Delta \vdash_{R} e_2 : T$ and $\Delta \Lleftarrow \{w : \forall \vec{\psi}.T\} \vdash_{R} e_1 : T'$ where $\vec{\psi} \cap FV(\Delta) = \varnothing$, then $\Delta \vdash_{R} e_1[w \backslash e_2] : T'$.*

Lemma A.2.3. *If $\Delta \vdash_{R} e : T$, then $\varphi\Delta \vdash_{R} e : \varphi T$ for any substitution φ.*

Lemma A.2.4 (Generalization). *Let $\Delta :: \{w : \sigma'\} \vdash_{R} e : T$ and $\sigma' \prec \sigma$. Then $\Delta :: \{w : \sigma\} \vdash_{R} e : A$.*

A.2.2 Transformation

Definition A.2.5. inst [KYC06] creates a monotype environment Γ from the polytype environment Δ, such that $\Gamma \prec \Delta$, by instantiating the contained polytypes using distinct renaming substitutions.

Lemma A.2.6. *We have the following properties:*

- $\Gamma \prec \Delta \iff [\![\Gamma]\!] \prec [\![\Delta]\!]$
- $\mathrm{GEN}_{\mathsf{inst}(\Delta)}(\Delta) = \Delta$
- $\mathrm{GEN}_{\Gamma}(\Delta) \prec \Delta$ *if* $\Gamma \prec \Delta$.

Lemma A.2.7. *Let e be a stage-n program and $m \geq n$. Then $FV([\![e]\!]_{R_0,\ldots,R_m}) \subseteq \bigcup_{i=m-n}^{m} FV(R_i)$.*

Proof. By a straightforward structural induction on e. □

Lemma A.2.8. *Let e be a stage-n λ^{gen}_{poly} expression with $FV(e) = \{x_1, \ldots, x_m\}$. Then,*

$$[\![e]\!]_{R_0,R_1,\ldots,R_n} = [\![e]\!]_{R'_0,R_1,\ldots,R_n}$$

if $R_0(x_i) = R'_0(x_i)$ for any $i \in \{1..m\}$.

Proof. By a straightforward structural induction on e. □

Corollary A.2.9. *Let e be a stage-0 expression with no free variables. Then $[\![e]\!]_{\{\}} = [\![e]\!]_{R_0}$ for any R_0.*

Lemma A.2.10. *Let e be a λ^{gen}_{poly} expression such that $e \in Val^{n+1}$. Then*

$$[\![e]\!]_{\{\},R_1,\ldots,R_{n+1}} = [\![e]\!]_{R_1,\ldots,R_{n+1}}$$

Proof. By a straightforward induction on the structure of e. □

Lemma A.2.11. *Let e_1 be a stage-n and e_2 a stage-0 λ^{gen}_{poly} expression with no free variables. Then*

$$[\![e_1]\!]_{R_0,R_1,\ldots,R_n}[z \backslash [\![e_2]\!]_{\{\}}] = [\![e_1[x \backslash e_2]^n]\!]_{R_0,R_1,\ldots,R_n}$$

where $R_0(x) = z$.

Proof. By induction on the structure of e_1. We only show the interesting cases. Other cases follow easily from the I.H.

- Case $e_1 = y$, $n > 0$: Because of our assumption on the renaming environments, $R_n(y) \neq z$. Hence, $[\![y]\!]_{R_0 \,\textsf{with}\, \{x=z\}, R_1, \ldots, R_n}[z \backslash [\![e_2]\!]_{\{\}}] = R_n(y)[z \backslash [\![e_2]\!]_{\{\}}] = R_n(y)$. And we have
  $[\![y[x \backslash e_2]^n]\!]_{R_0, R_1, \ldots, R_n} = [\![y]\!]_{R_0, R_1, \ldots, R_n} = R_n(y)$.
- Case $e_1 = x$, $n = 0$: We have $[\![x]\!]_{R_0 \,\textsf{with}\, \{x=z\}}[z \backslash [\![e_2]\!]_{\{\}}] = z[z \backslash [\![e_2]\!]_{\{\}}] = [\![e_2]\!]_{\{\}}$. We also have $[\![x[x \backslash e_2]^0]\!]_{R_0} = [\![e_2]\!]_{R_0}$. By Lemma A.2.8, $[\![e_2]\!]_{\{\}} = [\![e_2]\!]_{R_0}$.
- Case $e_1 = y$, $n = 0$, and $y \neq x$: Trivial. □

A.2.3 Relation Between Staged Programming and Record Calculus

Lemma A.2.12. *$R \cdot x \longrightarrow^*_\beta R(x)$ for any R.*

Proof. By structural induction on R.

- Case $R = \{\}$: We have $\{\} \cdot x \longrightarrow_\beta$ **error**. Also $\{\}(x) =$ **error** by definition.
- Case $R = r$: Trivial.
- Case $R = R' \,\textsf{with}\, \{y = z\}$: If $x = y$, then $R' \,\textsf{with}\, \{x = z\} \cdot x \longrightarrow_\beta z$ and $(R' \,\textsf{with}\, \{x = z\})(x) = z$.

  If $x \neq y$, then $R' \,\textsf{with}\, \{y = z\} \cdot x \longrightarrow_\beta R' \cdot x$ and $(R' \,\textsf{with}\, \{y = z\})(x) = R'(x)$. By I.H. we have $R' \cdot x \longrightarrow^*_\beta R'(x)$. □

Lemma A.2.13. *$(R(x))[r \backslash R'] \longrightarrow^*_\beta (R[r \backslash R'])(x)$ for any R, R'.*

Proof. By structural induction on R.

- Case $R = \{\}$: We have $\{\}(x)[r \backslash R'] =$ **error** and $(\{\}[r \backslash R'])(x) =$ **error** by definition.
- Case $R = r'$: If $r = r'$, then $(r(x))[r \backslash R'] = R' \cdot x$ and $(r[r \backslash R'])(x) = R'(x)$. By Lemma A.2.12, $R' \cdot x \longrightarrow^*_\beta R'(x)$.

  If $r \neq r'$, then $(r'(x))[r \backslash R'] = r' \cdot x$ and $(r'[r \backslash R'])(x) = r' \cdot x$.
- Case $R = R_1 \,\textsf{with}\, \{y = z\}$: If $x = y$, then $((R_1 \,\textsf{with}\, \{x = z\})(x))[r \backslash R'] = z$ and $((R_1 \,\textsf{with}\, \{x = z\})[r \backslash R'])(x) = z$.

  If $x \neq y$, then $((R_1 \,\textsf{with}\, \{y = z\})(x))[r \backslash R'] = (R_1(x))[r \backslash R']$ and $((R_1 \,\textsf{with}\, \{y = z\})[r \backslash R'])(x) = (R_1[r \backslash R'])(x)$. By I.H. we have $(R_1(x))[r \backslash R'] \longrightarrow^*_\beta (R_1[r \backslash R'])(x)$. □

Lemma A.2.14. *Let e be a stage-n λ^{gen}_{poly} expression. Then*

$$[\![e]\!]_{R_0, \ldots, R_n}[r_m \backslash R_m] \longrightarrow^*_\beta [\![e]\!]_{R_0[r_m \backslash R_m], \ldots, R_n[r_m \backslash R_m]}$$

Proof. By structural induction on e. In the VAR case we use Lemma A.2.13. In the BOX case we use the fact that the newly introduced environment variable R_{n+1} is fresh. Other cases easily follow from the I.H. □

Proof of Theorem 5.6.1. By induction on the structure of e_1, based on the last applied reduction rule. The proof mostly follows from the I.H. We only show interesting cases here.

- Case APP(3). We have

$$\frac{e' \in Val^0}{(\lambda x.e)e' \longrightarrow_0 e[x \backslash e']^0}$$

So,

$$\begin{aligned}
&[\![(\lambda x.e)e']\!]_{\{\}} = (\lambda z.[\![e]\!]_{\{x=z\}})[\![e']\!]_{\{\}} && \text{where } z \text{ is fresh}\\
&\longrightarrow_\beta [\![e]\!]_{\{x=z\}}[z \backslash [\![e']\!]_{\{\}}] &&\\
&= [\![e[x\backslash e']^0]\!]_{\{\}} && \text{by Lemma A.2.11}
\end{aligned}$$

- Case UBOX(2). We have

$$\frac{e \in Val^1}{\grave{}(\langle e\rangle) \longrightarrow_1 e}$$

Note that

$$\begin{aligned}
&[\![\grave{}(\langle e\rangle)]\!]_{\{\},R_1} = [\![\langle e\rangle]\!]_{\{\}}R_1 &&\\
&= (\lambda r_1.[\![e]\!]_{\{\},r_1})R_1 && \text{where } r_1 \text{ is fresh}\\
&\longrightarrow_\beta ([\![e]\!]_{\{\},r_1})[r_1 \backslash R_1] &&\\
&\longrightarrow^*_\beta [\![e]\!]_{\{\},R_1} && \text{by Lemma A.2.14}
\end{aligned}$$

- Case RUN(2). We have

$$\frac{e \in Val^1}{\mathsf{run}(\langle e\rangle) \longrightarrow_0 e}$$

So,

$$\begin{aligned}
&[\![\mathsf{run}(\langle e\rangle)]\!]_{\{\}} = ([\![\langle e\rangle]\!]_{\{\}})\{\} &&\\
&= (\lambda r_1.[\![e]\!]_{\{\},r_1})\{\} && \text{where } r_1 \text{ is fresh}\\
&\longrightarrow_\beta [\![e]\!]_{\{\},r_1}[r_1 \backslash \{\}] &&\\
&\longrightarrow^*_\beta [\![e]\!]_{\{\},\{\}} && \text{by Lemma A.2.14}\\
&= [\![e]\!]_{\{\}} && \text{by Lemma A.2.10}
\end{aligned}$$

- Case LIFT(2). We have

$$\frac{e \in Val^0}{\mathsf{lift}(e) \longrightarrow_0 \langle e\rangle}$$

So,

$$[\![\mathsf{lift}(e)]\!]_{\{\}} = (\lambda r_1.[\![e]\!]_{\{\}}) \qquad \text{where } r_1 \text{ is fresh}$$

Since $FV^0(e) = \varnothing$, $[\![e]\!]_{\{\}} = [\![e]\!]_{r_1}$ by Lemma A.2.8. Hence;

$$\begin{aligned} &= (\lambda r_1.[\![e]\!]_{r_1}) \\ &= (\lambda r_1.[\![e]\!]_{\{\},r_1}) && \text{by Lemma A.2.10} \\ &= [\![\langle e\rangle]\!]_{\{\}} \end{aligned}$$

□

Proof of Lemma 5.6.4. By structural induction on e. The proof mostly follows from the I.H. In UBOX, APP and RUN cases we use Lemma 5.6.3 and do reverse reasoning from types to expressions. We show the APP and UBOX cases.

- Case $e = e_1 e_2$ at stage n. We have, using Lemma 5.6.3,

$$\frac{\Delta \vdash_{\!R} [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : A \to B \quad \Delta \vdash_{\!R} [\![e_2]\!]_{\{\},R_1,\ldots,R_n} : A}{\Delta \vdash_{\!R} [\![e_1 e_2]\!]_{\{\},R_1,\ldots,R_n} : B}$$

  By I.H we have two subcases:

  1. $\exists e_1'$ such that $e_1 \longrightarrow_n e_1'$. In this case, $e_1 e_2 \longrightarrow_n e_1' e_2$.
  2. $e_1 \in Val^n$. By I.H. we have two subcases:
     (a) $\exists e_2'$ such that $e_2 \longrightarrow_n e_2'$. In this case, $e_1 e_2 \longrightarrow_n e_1 e_2'$.
     (b) $e_2 \in Val^n$. We again have two subcases:
        i. $n > 0$: In this case $e_1 e_2 \in Val^n$.
        ii. $n = 0$: Because $[\![e_1]\!]_{\{\}}$ has the function type $A \to B$ and e_1 is a value at stage-0, e_1 must be either $\lambda x.e_3$ or $\mathsf{fix}\, f(x).\, e_3$, for some e_3. Therefore we have either $(\lambda x.e_3)e_2 \longrightarrow_0 e_3[x\backslash e_2]^0$ or
        $(\mathsf{fix}\, f(x).\, e_3)e_2 \longrightarrow_0 e_3[f\backslash \mathsf{fix}\, f(x).\, e_3]^0[x\backslash e_2]^0$.

- Case $e =\, \grave{}(e_1)$ at stage $n+1$. Note that $[\![\grave{}(e_1)]\!]_{\{\},R_1,\ldots,R_{n+1}} = ([\![e_1]\!]_{\{\},R_1,\ldots,R_n})R_{n+1}$. We have

$$\Delta \vdash_{\!R} ([\![e_1]\!]_{\{\},R_1,\ldots,R_n})R_{n+1} : A$$

  Because the record expression R_{n+1} can only be given record types, we must have

$$\Delta \vdash_{\!R} [\![e_1]\!]_{\{\},R_1,\ldots,R_n} : \Gamma \to A$$

  for some Γ. By I.H. we have two subcases:

1. $\exists e_1'$ such that $e_1 \longrightarrow_n e_1'$. In this case, $\grave{}(e_1) \longrightarrow_{n+1} \grave{}(e_1')$.
2. $e_1 \in Val^n$. We have two subcases:
   (a) $n > 0$: In this case, $\grave{}(e_1) \in Val^{n+1}$.
   (b) $n = 0$: Recall that $e_1 \in Val^0$ and it types to $\Gamma \to A$. The only stage-0 value whose translation can have such a type is $\langle e' \rangle$ for some $e' \in Val^1$. Hence, $\grave{}(e_1) = \grave{}(\langle e' \rangle)$, and by ESUBOX, we have $\grave{}(\langle e' \rangle) \longrightarrow_1 e'$ □

Lemma A.2.15. *Let e be a stage-n λ^{gen}_{poly} expression. Then*

$$\Delta :: \{r_n : [\![\Delta_n]\!]\} \vdash_R [\![e]\!]_{R_0,\ldots,R_{n-1},R_n} : A$$

if and only if

$$\Delta :: \{r_n : [\![\Delta_n \lessdot\!+ \{x : \sigma\}]\!]\} \vdash_R [\![e]\!]_{R_0,\ldots,R_{n-1},R_n} : A$$

where $x \in dom(R_n)$.

Proof. This is the "weakening" lemma adapted to translation and records. Assume $R_n(x) = z$. Then, because of the translation, any occurrence of x at level n will be replaced with z and its type is grabbed from Δ — it is independent from r_n's type. Any variable $y \notin dom(R_n)$ will be translated to $r_n{\cdot}y$, and can still be given the same type because $[\![\Delta_n]\!](y) = [\![\Delta_n \lessdot\!+ \{x : \sigma\}]\!](y)$. □

Proof of Lemma 5.6.6. By structural induction on e.

- Case $e = c$.

  Trivial.

- Case $e = x$, ($\Longrightarrow$). We have $\Delta_0, \ldots, \Delta_n \vdash_S x : A$ with the premise $A \prec \Delta_n(x)$. Note that $[\![x]\!]_{R_0,\ldots,R_n} = R_n(x)$. We have two subcases.

  (i) Case $R_n(x) = z$ for some z: By the definition of type translation,

  $$([\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n})(z) = [\![\Delta_n(x)]\!]$$

  Using the fact that $[\![A]\!] \prec [\![\Delta_n(x)]\!]$, we get

  $$[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R z : [\![A]\!]$$

  (ii) Case $R_n(x) = r_n x$: By the definition of type translation, $([\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n})(r_n) = [\![\Delta_n]\!]$. Using the fact that $A \prec \Delta_n(x)$, it is easy to construct a Γ such that $\Gamma(x) = [\![A]\!]$ and $\Gamma \prec [\![\Delta_n]\!]$. Hence, $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R r_n : \Gamma$, which gives

  $$[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R r_n{\cdot}x : [\![A]\!]$$

- Case $e = x$, ($\Longleftarrow$). Note that $[\![x]\!]_{R_0,\ldots,R_n} = R_n(x)$. We have two subcases.

  (i) Case $R_n(x) = z$ for some z: We have

  $$[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R z : [\![A]\!]$$

  with the premise $[\![A]\!] \prec ([\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n})(z)$. By the definition of type translation, $([\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n})(z) = [\![\Delta_n(x)]\!]$, hence $[\![A]\!] \prec [\![\Delta_n(x)]\!]$. Therefore, $A \prec \Delta_n(x)$, which gives $\Delta_0, \ldots, \Delta_n \vdash_S x : A$.

  (ii) Case $R_n(x) = r_n{\cdot}x$: We have

  $$[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R r_n{\cdot}x : [\![A]\!]$$

  with the premises $\Gamma \prec [\![\Delta_n]\!]$ and $\Gamma(x) = [\![A]\!]$. These two premises imply $A \prec \Delta_n(x)$, which gives $\Delta_0, \ldots, \Delta_n \vdash_S x : A$.

- Case $e = \lambda x.e'$, ($\Longrightarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n} = \lambda z.\, [\![e']\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}}$ where z is fresh.

  1. We have $\Delta_0, \ldots, \Delta_n \vdash_S \lambda x.e' : A \to B$ with the premise
  2. $\Delta_0, \ldots, \Delta_n {<\!\!+} \{x : A\} \vdash_S e' : B$.
  3. $[\![\Delta_0, \ldots, \Delta_n {<\!\!+} \{x : A\}]\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} \vdash_R [\![e']\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} : [\![B]\!]$ by I.H. and (2).
  4. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} {<\!\!+} \{z : [\![A]\!]\} \vdash_R [\![e']\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} : [\![B]\!]$ by (3) and Lemma A.2.15.
  5. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda z.[\![e']\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} : [\![A]\!] \to [\![B]\!]$ by (4) and TRABS.

- Case $e = \lambda x.e'$, ($\Longleftarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n} = \lambda z.\, [\![e']\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}}$ where z is fresh.

  1. We have $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda z.[\![e']\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} : [\![C]\!]$
  2. By TRABS, we have $[\![C]\!] = A' \to B'$ for some A', B'. By Lemma 5.5.2, there exist A, B such that $[\![A]\!] = A'$ and $[\![B]\!] = B'$. Therefore $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda z.[\![e']\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} : [\![A]\!] \to [\![B]\!]$
  3. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} {<\!\!+} \{z : [\![A]\!]\} \vdash_R [\![e']\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} : [\![B]\!]$ as a premise of by (2).
  4. $[\![\Delta_0, \ldots, \Delta_n {<\!\!+} \{x : A\}]\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} \vdash_R [\![e']\!]_{R_0,\ldots,R_n \,\mathsf{with}\, \{x=z\}} : [\![B]\!]$ by (3) and Lemma A.2.15.
  5. $\Delta_0, \ldots, \Delta_n {<\!\!+} \{x : A\} \vdash_S e' : B$ by I.H. and (4).
  6. $\Delta_0, \ldots, \Delta_n \vdash_S \lambda x.e' : A \to B$ (5) and TSABS.

- Case $e = \lambda^* x.e'$ is very similar to the abstraction case.
- Case $e = \mathsf{fix}\, f(x).\, e'$ is very similar to the abstraction case.
- Case $e = e_1 e_2$, ($\Longrightarrow$). Easily follows from the I.H.
- Case $e = e_1 e_2$, ($\Longleftarrow$).

  1. We have $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_1 e_2]\!]_{R_0,\ldots,R_n} : [\![A]\!]$ with the premises
  2. $\quad [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_1]\!]_{R_0,\ldots,R_n} : T \rightarrow [\![A]\!]$and
  3. $\quad [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_2]\!]_{R_0,\ldots,R_n} : T$ for some T
  4. $T = B' \in RLegType$ by Lemma 5.6.3
  5. $B' = [\![B]\!]$ for some B by Lemma 5.5.2
  6. $\Delta_0, \ldots, \Delta_n \vdash_S e_2 : B$ by (3), (5), and I.H.
  7. $\Delta_0, \ldots, \Delta_n \vdash_S e_1 : B \rightarrow A$ by (2), (5), and I.H.
  8. $\Delta_0, \ldots, \Delta_n \vdash_S e_1 e_2 : A$ by (6), (7), and TSAPP.

- Case $e = \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ uses the same principles in the abstraction and application cases together with the fact that type translation does not alter bound/unbound type variables.
- Case $e = \langle e' \rangle$, ($\Longrightarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n} = \lambda r_{n+1}.\, [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}}$ where r_{n+1} is fresh.

  1. We have $\Delta_0, \ldots, \Delta_n \vdash_S \langle e' \rangle : \Box(\Gamma \triangleright A)$ with the premise
  2. $\quad \Delta_0, \ldots, \Delta_n, \Gamma \vdash_S e' : A$.
  3. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r_{n+1}} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![A]\!]$ by I.H. and (2).
  4. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r_{n+1}} = [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \Leftarrow\!\!+ \{r_{n+1} : [\![\Gamma]\!]\}$ by the definition of type translation
  5. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \Leftarrow\!\!+ \{r_{n+1} : [\![\Gamma]\!]\} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![A]\!]$ by (3) and (4).
  6. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda r_{n+1}.[\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![\Gamma]\!] \rightarrow [\![A]\!]$ by (5) and TRABS.

- Case $e = \langle e' \rangle$, ($\Longleftarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n} = \lambda r_{n+1}.\, [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}}$ where r_{n+1} is fresh.

  1. We have $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda r_{n+1}.[\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![A]\!]$
  2. $[\![A]\!] = \Gamma' \rightarrow B'$ for some Γ' and B'. by TRABS
  3. $[\![\Gamma]\!] = \Gamma'$ and $[\![B]\!] = B'$ for some Γ and B. by Lemma 5.5.2
  4. (1) has the premise
     $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \Leftarrow\!\!+ \{r_{n+1} : [\![\Gamma]\!]\} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![B]\!]$ by TRABS

5. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r_{n+1}} = [\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \lessdot\!\!+ \{r_{n+1} : [\![\Gamma]\!]\}$ by the definition of type translation
6. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r_{n+1}} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r_{n+1}} : [\![B]\!]$ by (4) and (5).
7. $\Delta_0, \ldots, \Delta_n, \Gamma \vdash_S e' : B$ by I.H. and (6).
8. $\Delta_0, \ldots, \Delta_n \vdash_S \langle e' \rangle : \Box(\Gamma \triangleright B)$ by (7) and TSBOX.

- Case $e = \,`(e_1)$, ($\Longrightarrow$). Note that $[\![e]\!]_{R_0,\ldots,R_n,R_{n+1}} = ([\![e_1]\!]_{R_0,\ldots,R_n})R_{n+1}$.

1. We have $\Delta_0, \ldots, \Delta_n, \Delta_{n+1} \vdash_S \,`(e_1) : A$ with the premises
2. $\quad\Delta_0, \ldots, \Delta_n \vdash_S e_1 : \Box(\Gamma \triangleright A)$ and
3. $\quad\Gamma \prec \Delta_{n+1}$.
4. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_1]\!]_{R_0,\ldots,R_n} : [\![\Gamma]\!] \to [\![A]\!]$ by I.H. and (2).
5. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R [\![e_1]\!]_{R_0,\ldots,R_n} : [\![\Gamma]\!] \to [\![A]\!]$ by (4) and Lemma A.2.1.
6. Without loss of generality, assume $R_{n+1} = r_{n+1}$ with $\{x = z\}$. We have
7. $\quad([\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}})(r_{n+1}) = [\![\Delta_{n+1}]\!]$
8. From (3) we have $[\![\Gamma]\!] \prec [\![\Delta_{n+1}]\!]$.
9. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R r_{n+1} : [\![\Gamma]\!]$ by (7), (8), and TRVAR.
10. $([\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}})(z) = [\![\Delta_{n+1}(x)]\!] = [\![\Delta_{n+1}]\!](x)$ by definition
11. From (8) we have $[\![\Gamma]\!](x) \prec [\![\Delta_{n+1}]\!](x)$.
12. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R z : [\![\Gamma]\!](x)$ by (10), (11), and TRVAR.
13. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R R_{n+1} : [\![\Gamma]\!] \lessdot\!\!+ \{x : [\![\Gamma]\!](x)\}$ by (9), (12), and TRUPD.
14. $[\![\Gamma]\!] \lessdot\!\!+ \{x : [\![\Gamma]\!](x)\} = [\![\Gamma]\!]$
15. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R R_{n+1} : [\![\Gamma]\!]$ by (13) and (14).
16. $[\![\Delta_0, \ldots, \Delta_n, \Delta_{n+1}]\!]_{R_0,\ldots,R_n,R_{n+1}} \vdash_R ([\![e_1]\!]_{R_0,\ldots,R_n})R_{n+1} : [\![A]\!]$ by (5), (15) and TRAPP.

- Case $e = \,`(e_1)$, ($\Longleftarrow$). This case applies the ($\Longrightarrow$) case in the backwards direction with the additional use of Lemma 5.5.2 and the fact that $\Gamma \prec \Delta$ and $A \prec \Delta(x)$ imply $\Gamma \lessdot\!\!+ \{x : A\} \prec \Delta$.

- Case $e = \mathsf{run}(e')$.

  Follows easily from the I.H.

- Case $e = \mathsf{lift}(e')$.

  Follows easily from the I.H. and Lemma A.2.1. □

A.2.4 Extension with Pluggable Declarations

Proof of Theorem 5.8.1. By structural induction on e_1, based on the last applied reduction. We only show the cases for the new syntax.

- Case $e_1 = \langle\rangle$. Not possible because $\langle\rangle \in Val^n$.
- Case $e_1 = \langle x = e\rangle$. We have

$$\frac{e \longrightarrow_{n+1} e'}{\langle x = e\rangle \longrightarrow_n \langle x = e'\rangle}$$

  and

$$\frac{\varnothing, \Delta_1, \ldots, \Delta_n, \Gamma \vdash_P e : A}{\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P \langle x = e\rangle : \Diamond(\Gamma \triangleright \Gamma \lessdot\!\!+ \{x : A\})}$$

  By I.H., we have $\varnothing, \Delta_1, \ldots, \Delta_n, \Gamma \vdash_P e' : A$, which gives, by TSDEC, that

$$\varnothing, \Delta_1 \ldots, \Delta_n \vdash_P \langle x = e'\rangle : \Diamond(\Gamma \triangleright \Gamma \lessdot\!\!+ \{x : A\})$$

- Case $e_1 = \mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4$. The two cases when we have

$$\frac{e_3 \longrightarrow_n e_3'}{\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \grave{}(e_3')\ \mathsf{in}\ e_4}$$

  and

$$\frac{e_3 \in Val^n \quad e_4 \longrightarrow_{n+1} e_4'}{\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4'}$$

  easily follow from the I.H.

- Case $e_1 = \mathsf{let}\ \grave{}(\langle x = e_3\rangle)\ \mathsf{in}\ e_4$ and

$$\frac{e_3 \in Val^1 \quad e_4 \in Val^1}{\mathsf{let}\ \grave{}(\langle x = e_3\rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 \mathsf{let}\ x = e_3\ \mathsf{in}\ e_4}$$

  We have

$$\frac{\varnothing \vdash_P \langle x = e_3\rangle : \Diamond(\Gamma \triangleright \Gamma') \quad \Gamma \prec \Delta_1 \quad \varnothing, \Gamma' \vdash_P e_4 : A}{\varnothing, \Delta_1 \vdash_P \mathsf{let}\ \grave{}(\langle x = e_3\rangle)\ \mathsf{in}\ e_4 : A}$$

  By TSDEC, for some B, we must have $\varnothing, \Gamma \vdash_P e_3 : B$ and $\Gamma' = \Gamma \lessdot\!\!+ \{x : B\}$.

  By the Generalization Lemma (from [KYC06]) and the premise $\Gamma \prec \Delta_1$, we have $\varnothing, \Delta_1 \vdash_P e_3 : B$.

  Again by the Generalization Lemma and the fact that $\Gamma \lessdot\!\!+ \{x : B\} \prec \Delta_1 \lessdot\!\!+ \{x : \mathrm{GEN}_B(\varnothing, \Delta_1)\}$, we have $\varnothing, \Delta_1 \lessdot\!\!+ \{x : \mathrm{GEN}_B(\varnothing, \Delta_1)\} \vdash_P e_4 : A$.

  Finally, by TSLET, we obtain $\varnothing, \Delta_1 \vdash_P \mathsf{let}\ x = e_3\ \mathsf{in}\ e_4 : A$.

- Case $e_1 = \mathsf{let}\ \grave{}(\langle\rangle)\ \mathsf{in}\ e$ is straightforward. $\square$

Proof of Theorem 5.8.2. By structural induction on e_1. We only show the cases for the new syntax.

- Case $e_1 = \langle\rangle$. $\langle\rangle \in Val^n$.

- Case $e_1 = \langle x = e\rangle$. We have

$$\frac{\varnothing, \Delta_1, \ldots, \Delta_n, \Gamma \vdash_P e : A}{\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P \langle x = e\rangle : \Diamond(\Gamma \triangleright \Gamma \lessdot\!\!+ \{x : A\})}$$

  By I.H. we either have $e \in Val^{n+1}$, which means $\langle x = e\rangle \in Val^n$, or we have e' such that $e \longrightarrow_{n+1} e'$, which means $\langle x = e\rangle \longrightarrow_n \langle x = e'\rangle$.

- Case $e_1 = \mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4$. We have

$$\frac{\varnothing, \Delta_1, \ldots, \Delta_n \vdash_P e_3 : \Diamond(\Gamma \triangleright \Gamma') \quad \Gamma \prec \Delta_{n+1} \quad \varnothing, \Delta_1, \ldots, \Delta_n, \Gamma' \vdash_P e_4 : A}{\varnothing, \Delta_1, \ldots, \Delta_n, \Delta_{n+1} \vdash_P \mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 : A}$$

  By I.H. we either have $e_3 \longrightarrow_n e_3'$, which means $\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \grave{}(e_3')\ \mathsf{in}\ e_4$. Or we have $e_3 \in Val^n$. In this case, by I.H. we have two subcases:

  - $e_4 \longrightarrow_{n+1} e_4'$, which means $\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4'$.
  - $e_4 \in Val^{n+1}$. We again have two subcases. If $n > 0$, we have $\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 \in Val^{n+1}$. If $n = 0$, we first recall that $e_3 \in Val^0$ and that e_3 types to $\Diamond(\Gamma \triangleright \Gamma')$. The only stage-0 value that can be given such a type is either $\langle x = e_5\rangle$ for some $e_5 \in Val^1$, which by ESLET2 gives $\mathsf{let}\ \grave{}(\langle x = e_5\rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 \mathsf{let}\ x = e_5\ \mathsf{in}\ e_4$; or $\langle\rangle$, which again by ESLET2 gives $\mathsf{let}\ \grave{}(\langle\rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 e_4$. □

Lemma A.2.16. *Let e be a λ^{decl}_{poly} expression such that $e \in Val^{n+1}$. Then $\delta(e)$ is a λ^{gen}_{poly} expression such that $\delta(e) \in Val^{n+1}$.*

Proof. By a straightforward structural induction on e. □

Proof of Theorem 5.8.3. By structural induction on e_1, based on the last applied reduction. The proof mostly follows from the I.H. We show the most interesting cases here.

- We have

$$\frac{e \longrightarrow_{n+1} e'}{\langle x = e\rangle \longrightarrow_n \langle x = e'\rangle}$$

  By I.H., $\delta(e) \longrightarrow^*_{n+1} \delta(e')$. Hence,

$$\begin{aligned} \delta(\langle x = e\rangle) &= (\lambda v.\lambda y.\langle \mathsf{let}\ x = \grave{}(v)\ \mathsf{in}\ \grave{}(y)\rangle)\ \langle \delta(e)\rangle \\ &\longrightarrow^*_n (\lambda v.\lambda y.\langle \mathsf{let}\ x = \grave{}(v)\ \mathsf{in}\ \grave{}(y)\rangle)\ \langle \delta(e')\rangle \\ &= \delta(\langle x = e'\rangle) \end{aligned}$$

- We have
$$\frac{e_3 \in Val^1 \quad e_4 \in Val^1}{\mathsf{let}\ \grave{}(\langle x = e_3\rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 \mathsf{let}\ x = e_3\ \mathsf{in}\ e_4}$$

  Note that
$$\delta(\mathsf{let}\ \grave{}(\langle x = e_3\rangle)\ \mathsf{in}\ e_4) \quad = \quad \grave{}(((\lambda v.\lambda y.\langle \mathsf{let}\ x = \ \grave{}(v)\ \mathsf{in}\ \grave{}(y)\rangle)\ \langle\delta(e_3)\rangle)\ \langle\delta(e_4)\rangle)$$

  By Lemma A.2.16, $\delta(e_3), \delta(e_4) \in Val^1$. Hence,
$$\begin{aligned}
&\grave{}(((\lambda v.\lambda y.\langle \mathsf{let}\ x = \ \grave{}(v)\ \mathsf{in}\ \grave{}(y)\rangle)\ \langle\delta(e_3)\rangle)\ \langle\delta(e_4)\rangle)\\
&\longrightarrow_1 \grave{}((\lambda y.\langle \mathsf{let}\ x = \ \grave{}(\langle\delta(e_3)\rangle)\ \mathsf{in}\ \grave{}(y)\rangle)\ \langle\delta(e_4)\rangle)\\
&\longrightarrow_1 \grave{}(\langle \mathsf{let}\ x = \ \grave{}(\langle\delta(e_3)\rangle)\ \mathsf{in}\ \grave{}(\langle\delta(e_4)\rangle)\rangle)\\
&\longrightarrow_1 \grave{}(\langle \mathsf{let}\ x = \delta(e_3)\ \mathsf{in}\ \grave{}(\langle\delta(e_4)\rangle)\rangle)\\
&\longrightarrow_1 \grave{}(\langle \mathsf{let}\ x = \delta(e_3)\ \mathsf{in}\ \delta(e_4)\rangle)\\
&\longrightarrow_1 \mathsf{let}\ x = \delta(e_3)\ \mathsf{in}\ \delta(e_4)\\
&= \delta(\mathsf{let}\ x = e_3\ \mathsf{in}\ e_4)
\end{aligned}$$

$\square$

Proof of Theorem 5.8.4. By structural induction on e. The most interesting cases are below.

- We have
$$\frac{\Delta_0, \ldots, \Delta_n, \Gamma \vdash_P e_1 : A}{\Delta_0, \ldots, \Delta_n \vdash_P \langle x = e_1\rangle : \Diamond(\Gamma \triangleright \Gamma {<\!\!+} \{x : A\})}$$

  Note that
$$\begin{aligned}
\delta(\langle x = e_1\rangle) \quad &= \quad (\lambda v.\lambda y.\langle \mathsf{let}\ x = \ \grave{}(v)\ \mathsf{in}\ \grave{}(y)\rangle)\ \langle\delta(e_1)\rangle\\
\delta(\Diamond(\Gamma \triangleright \Gamma {<\!\!+} \{x : A\})) \quad &= \quad \Box(\delta(\Gamma {<\!\!+} \{x : A\}) \triangleright B) \to \Box(\delta(\Gamma) \triangleright B) \text{ for any } B
\end{aligned}$$

  We now proceed as follows:

  1. $\delta(\Delta_0), \ldots, \delta(\Delta_n), \delta(\Gamma) \vdash_S \delta(e_1) : \delta(A)$ by I.H.
  2. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \langle\delta(e_1)\rangle : \Box(\delta(\Gamma) \triangleright \delta(A))$ by (1) and TSBOX
  3. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S (\lambda v.\lambda y.\langle \mathsf{let}\ x = \ \grave{}(v)\ \mathsf{in}\ \grave{}(y)\rangle) : \Box(\delta(\Gamma) \triangleright \delta(A)) \to \Box(\delta(\Gamma {<\!\!+} \{x : A\}) \triangleright B) \to \Box(\delta(\Gamma) \triangleright B)$ by a series of typing rules. Note that this judgment can be derived for any B.
  4. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S (\lambda v.\lambda y.\langle \mathsf{let}\ x = \ \grave{}(v)\ \mathsf{in}\ \grave{}(y)\rangle)\ \langle\delta(e_1)\rangle : \Box(\delta(\Gamma {<\!\!+} \{x : A\}) \triangleright B) \to \Box(\delta(\Gamma) \triangleright B)$ by (2), (3), and TSAPP

- We have

$$\frac{\Delta_0, \ldots, \Delta_n \vdash_P e_1 : \Diamond(\Gamma \rhd \Gamma') \quad \Gamma \prec \Delta_{n+1} \qquad \Delta_0, \ldots, \Delta_n, \Gamma' \vdash_P e_2 : A}{\Delta_0, \ldots, \Delta_n, \Delta_{n+1} \vdash_P \mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2 : A}$$

  Note that

$$\delta(\mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2) \quad = \quad \grave{}(\delta(e_1)\ \langle\delta(e_2)\rangle)$$

  We now proceed as follows:

  1. $\delta(\Diamond(\Gamma \rhd \Gamma')) = \Box(\delta(\Gamma') \rhd \delta(A)) \to \Box(\delta(\Gamma) \rhd \delta(A))$ by the definition of $\delta(\cdot)$
  2. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \delta(e_1) : \Box(\delta(\Gamma') \rhd \delta(A)) \to \Box(\delta(\Gamma) \rhd \delta(A))$ by (1) and I.H.
  3. $\delta(\Delta_0), \ldots, \delta(\Delta_n), \delta(\Gamma') \vdash_S \delta(e_2) : \delta(A)$ by I.H.
  4. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \langle\delta(e_2)\rangle : \Box(\delta(\Gamma') \rhd \delta(A))$ by (3) and TSBOX
  5. $\delta(\Delta_0), \ldots, \delta(\Delta_n) \vdash_S \delta(e_1)\ \langle\delta(e_2)\rangle : \Box(\delta(\Gamma) \rhd \delta(A))$ by (2), (4) and TSAPP
  6. $\delta(\Gamma) \prec \delta(\Delta_{n+1})$ by the premise of the assumption
  7. $\delta(\Delta_0), \ldots, \delta(\Delta_n), \delta(\Delta_{n+1}) \vdash_S \grave{}(\delta(e_1)\ \langle\delta(e_2)\rangle) : \delta(A)$ by (5), (6) and TSUBOX

$\Box$

Note that the extension with pluggable declarations to the translation preserves the Lemmata A.2.7 and 5.6.3.

Proof of Theorem 5.8.5. By induction on the structure of e_1, based on the last applied reduction rule. This proof is an extension of Theorem 5.6.1 with the pluggable declaration syntax. The cases mostly follow from the I.H. We only show the most interesting case. Note that the extension with pluggable declarations preserves Lemmata A.2.8, A.2.10, A.2.11, and A.2.14, which are used in the proof of Theorem 5.6.1 (and here in this proof, too).

- Case LET2(3). We have

$$\frac{e_3 \in Val^1 \quad e_4 \in Val^1}{\mathsf{let}\ \grave{}(\langle x = e_3\rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 \mathsf{let}\ x = e_3\ \mathsf{in}\ e_4}$$

Note that

$$
\begin{aligned}
&[\![\mathsf{let}\ \grave{}(\langle x = e_3\rangle)\ \mathsf{in}\ e_4]\!]_{\{\},R_1} \\
&= ([\![\langle x = e_3\rangle]\!]_{\{\}})\kappa\,(\lambda r.[\![e_4]\!]_{\{\},r})\,R_1 \\
&= ([\![\langle x = e_3\rangle]\!]_{\{\}})\kappa\,(\lambda r.[\![e_4]\!]_{\{\},r})\,R_1 \\
&= (\lambda\kappa.\lambda y.\lambda r.\ \mathsf{let}\ z = [\![e_3]\!]_{\{\},r}\ \mathsf{in}\ y(r\ \mathsf{with}\ \{x = z\}))\kappa\,(\lambda r.[\![e_4]\!]_{\{\},r})R_1 \\
&\longrightarrow_\beta (\lambda y.\lambda r.\ \mathsf{let}\ z = [\![e_3]\!]_{\{\},r}\ \mathsf{in}\ y(r\ \mathsf{with}\ \{x = z\}))(\lambda r.[\![e_4]\!]_{\{\},r})R_1 \\
&\longrightarrow_\beta (\lambda r.\ \mathsf{let}\ z = [\![e_3]\!]_{\{\},r}\ \mathsf{in}\ (\lambda r.[\![e_4]\!]_{\{\},r})(r\ \mathsf{with}\ \{x = z\}))R_1 \\
&\longrightarrow_\beta (\mathsf{let}\ z = [\![e_3]\!]_{\{\},r}[r\backslash R_1]\ \mathsf{in}\ (\lambda r.[\![e_4]\!]_{\{\},r})(R_1\ \mathsf{with}\ \{x = z\})) \\
&\longrightarrow_\beta \mathsf{let}\ z = [\![e_3]\!]_{\{\},r}[r\backslash R_1]\ \mathsf{in}\ [\![e_4]\!]_{\{\},r}[r\backslash R_1\ \mathsf{with}\ \{x = z\}] \\
&\longrightarrow_\beta^* \mathsf{let}\ z = [\![e_3]\!]_{\{\},R_1}\ \mathsf{in}\ [\![e_4]\!]_{\{\},R_1\ \mathsf{with}\ \{x=z\}} \qquad \text{by Lemma A.2.14} \\
&= [\![\mathsf{let}\ x = e_3\ \mathsf{in}\ e_4]\!]_{\{\},R_1}
\end{aligned}
$$

□

Proof of Theorem 5.8.7. By structural induction on e_1. The proof is the same as Theorem 5.6.4, except it is extended for the new syntax for pluggable declarations. The proof for the new cases mostly follow easily from the I.H. We only show the most interesting case here.

Let e_1 be the stage-$n+1$ expression $\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4$. We have

$$\Delta \vdash_R [\![\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4]\!]_{\{\},R_1,\ldots,R_{n+1}} : A$$

Note that

$$[\![\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4]\!]_{\{\},R_1,\ldots,R_{n+1}} = ([\![e_3]\!]_{\{\},R_1,\ldots,R_n})\kappa\,(\lambda r.[\![e_4]\!]_{\{\},R_1,\ldots,R_n,r})\,R_{n+1}$$

As the (sub)premises, we must have

$$\Delta \vdash_R R_{n+1} : \Gamma$$

$$\Delta \vdash_R (\lambda r.[\![e_4]\!]_{\{\},R_1,\ldots,R_n,r}) : \Gamma' \to B$$

$$\Delta \vdash_R [\![e_3]\!]_{\{\},R_1,\ldots,R_n} : \kappa \to (\Gamma' \to B) \to \Gamma \to A$$

for some Γ, Γ', and B. As the premise of the second judgment above, we also must have

$$\Delta {<\!\!+} \{r : \Gamma'\} \vdash_R [\![e_4]\!]_{\{\},R_1,\ldots,R_n,r} : B$$

By I.H. we have two subcases:

- $\exists e_3'$ such that $e_3 \longrightarrow_n e_3'$. In this case, $\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \grave{}(e_3')\ \mathsf{in}\ e_4$.

- $e_3 \in Val^n$. In this case, by I.H., we have two subcases:
  - $\exists e_4'$ such that $e_4 \longrightarrow_{n+1} e_4'$. In this case, $\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 \longrightarrow_{n+1} \mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4'$.
  - $e_4 \in Val^n$. We again have two subcases:
    (i) $n > 0$: In this case, $\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 \in Val^{n+1}$.
    (ii) $n = 0$: Recall that $e_3 \in Val^0$ and its translation types to $\kappa \to (\Gamma' \to B) \to \Gamma \to A$. The only stage-0 value whose translation can have such a type is $\langle x = e'' \rangle$ for some x and $e'' \in Val^1$. Hence, $\mathsf{let}\ \grave{}(e_3)\ \mathsf{in}\ e_4 = \mathsf{let}\ \grave{}(\langle x = e'' \rangle)\ \mathsf{in}\ e_4$, and by ESLET2, we have

$$\mathsf{let}\ \grave{}(\langle x = e'' \rangle)\ \mathsf{in}\ e_4 \longrightarrow_1 \mathsf{let}\ x = e''\ \mathsf{in}\ e_4 \qquad \square$$

Proof of Theorem 5.8.9. By structural induction on e. The proof is the same as Theorem 5.6.6, except being extended for the new syntax for pluggable declarations. The proof for the new cases mostly follow easily from the I.H. We show the two interesting cases here. Note that the extension with pluggable declarations preserves Lemma A.2.15, which is used in the proof of Theorem 5.6.6 (and here in this proof, too).

- Case $e = \langle x = e' \rangle$ at stage n. Note that

$$[\![\langle x = e' \rangle]\!]_{R_0,\ldots,R_n} = \lambda\kappa.\lambda y.\lambda r.\ \mathsf{let}\ z = [\![e']\!]_{R_0,\ldots,R_n,r}\ \mathsf{in}\ y(r\ \mathsf{with}\ \{x = z\})$$

  where r, y, z are fresh.

  1. We have $\Delta_0, \ldots, \Delta_n \vdash_P \langle x = e' \rangle : \Diamond(\Gamma \triangleright \Gamma {<\!\!+} \{x : A\})$
  2. Note that $[\![\Diamond(\Gamma \triangleright \Gamma {<\!\!+} \{x : A\})]\!] = \kappa \to ([\![\Gamma]\!] {<\!\!+} \{x : [\![A]\!]\} \to B) \to ([\![\Gamma]\!] \to B)$ for any B.
  3. $\Delta_0, \ldots, \Delta_n, \Gamma \vdash_P e' : A$ — as a premise of (1)
  4. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r} \vdash_R [\![e']\!]_{R_0,\ldots,R_n,r} : [\![A]\!]$ — by I.H. and (3)
  5. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r} \vdash_R r : [\![\Gamma]\!]$ — by TRVAR
  6. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r} {<\!\!+} \{z : \mathrm{GEN}_{[\![A]\!]}(\ldots)\} \vdash_R (r\ \mathsf{with}\ \{x = z\}) : [\![\Gamma]\!] {<\!\!+} \{x : [\![A]\!]\}$
     by (4), (5), and TRUPD
  7. $[\![\Delta_0, \ldots, \Delta_n, \Gamma]\!]_{R_0,\ldots,R_n,r} {<\!\!+} \{z : \mathrm{GEN}_{[\![A]\!]}(\ldots)\} \vdash_R$
     $\lambda\kappa.\lambda y.\lambda r.\ \mathsf{let}\ z = [\![e']\!]_{R_0,\ldots,R_n,r}\ \mathsf{in}\ y(r\ \mathsf{with}\ \{x = z\}) :$
     $\kappa \to ([\![\Gamma]\!] {<\!\!+} \{x : [\![A]\!]\} \to B) \to ([\![\Gamma]\!] \to B)$
     by (6), TRLET, and multiple applications of TRABS

- Case $e = \mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2$ at stage $n+1$. Note that

$$[\![\mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2]\!]_{R_0,\ldots,R_{n+1}} = ([\![e_1]\!]_{R_0,\ldots,R_n})\kappa\,(\lambda r.[\![e_2]\!]_{R_0,\ldots,R_n,r})\,R_{n+1}$$

where r is fresh.

1. We have $\Delta_0, \ldots, \Delta_{n+1} \vdash_P \mathsf{let}\ \grave{}(e_1)\ \mathsf{in}\ e_2 : A$
2. $\Delta_0, \ldots, \Delta_n \vdash_P e_1 : \Diamond(\Gamma_1 \triangleright \Gamma_2)$ for some Γ_1, Γ_2, as a premise of (1)
3. $\Gamma_1 \prec \Delta_{n+1}$ as a premise of (1)
4. $\Delta_0, \ldots, \Delta_n, \Gamma_2 \vdash_P e_2 : A$ as a premise of (1)
5. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R [\![e_1]\!]_{R_0,\ldots,R_n} : \kappa \to ([\![\Gamma_2]\!] \to [\![A]\!]) \to [\![\Gamma_1]\!] \to [\![A]\!]$ by I.H. and (2)
6. $[\![\Delta_0, \ldots, \Delta_n, \Gamma_2]\!]_{R_0,\ldots,R_n,r} \vdash_R [\![e_2]\!]_{R_0,\ldots,R_n,r} : [\![A]\!]$ by I.H. and (4)
7. $[\![\Delta_0, \ldots, \Delta_n]\!]_{R_0,\ldots,R_n} \vdash_R \lambda r.[\![e_2]\!]_{R_0,\ldots,R_n,r} : [\![\Gamma_2]\!] \to [\![A]\!]$ by (6) and TRABS
8. $[\![\Gamma_1]\!] \prec [\![\Delta_{n+1}]\!]$ by (3) and Lemma A.2.6
9. $[\![\Delta_0, \ldots, \Delta_{n+1}]\!]_{R_0,\ldots,R_{n+1}} \vdash_R R_{n+1} : [\![\Gamma_1]\!]$
   by (8) and multiple TRVAR (see Theorem 5.6.6, case $e = \grave{}(e_1)$, ($\Longrightarrow$), items (6) through (15) for a similar case)
10. $[\![\Delta_0, \ldots, \Delta_{n+1}]\!]_{R_0,\ldots,R_{n+1}} \vdash_R ([\![e_1]\!]_{R_0,\ldots,R_n})\kappa\,(\lambda r.[\![e_2]\!]_{R_0,\ldots,R_n,r})\,R_{n+1} : [\![A]\!]$
    by (5), (7), (9) and TRAPP

□

A.2.5 Extension with References

Lemma A.2.17. *If $\Sigma' \supseteq \Sigma$, then $\Sigma; \Delta \vdash_R e : A \implies \Sigma'; \Delta \vdash_R e : A$.*

Proof. This is a standard lemma. □

Lemma A.2.18. *Let $[\![e]\!]_{R_0,\ldots,R_n} = (e_0, \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$ for some stage-n expression e. Then,*

$$FV(e_0) \subseteq FV(R_n) \cup \{\vec{\pi_1}\}$$

$$FV(\vec{e_1}) \subseteq FV(R_{n-1}) \cup \{\vec{\pi_2}\}$$

$$\vdots$$

$$FV(\vec{e}_{m-1}) \subseteq FV(R_{n-m+1})\{\vec{\pi_m}\}$$

$$FV(\vec{e_m}) \subseteq FV(R_{n-m})$$

Proof. By structural induction on e. We show the quotation and antiquotation cases. Other cases are straightforward from the I.H.

- Case $e = \langle e' \rangle$, stage n.

  1. Suppose $[\![e']\!]_{R_0,\ldots,R_n,r} = (e'', \{(\vec{\pi_0}, \vec{e_0})\} :: \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$

2. By I.H.

$$FV(e'') \subseteq \{r\} \cup \{\vec{\pi_0}\}$$

$$FV(\vec{e_0}) \subseteq FV(R_n) \cup \{\vec{\pi_1}\}$$

$$FV(\vec{e_1}) \subseteq FV(R_{n-1}) \cup \{\vec{\pi_2}\}$$

$$\vdots$$

$$FV(\vec{e}_{m-1}) \subseteq FV(R_{n-m+1}) \cup \{\vec{\pi_m}\}$$

$$FV(\vec{e_m}) \subseteq FV(R_{n-m}\}$$

3. By definition, $[\![\langle e' \rangle]\!]_{R_0,\ldots,R_n} = ((\lambda\vec{\pi_0}.\lambda r.\, e'')\vec{e_0}, \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$
4. $FV((\lambda\vec{\pi_0}.\lambda r.\, e'')\vec{e_0}) \subseteq (\{r\} \cup \{\vec{\pi_0}\} \setminus \{\vec{\pi_0}, r\}) \cup FV(R_n) \cup \{\vec{\pi_1}\} = FV(R_n) \cup \{\vec{\pi_1}\}$
5. Properties for $FV(\vec{e_1}), \ldots, FV(\vec{e}_m)$ are immediate from the I.H.

- Case $e = \,`(e')$, stage $n+1$.

1. Suppose $[\![e']\!]_{R_0,\ldots,R_n} = (e'', \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$
2. By I.H.

$$FV(e'') \subseteq FV(R_n) \cup \{\vec{\pi_1}\}$$

$$FV(\vec{e_1}) \subseteq FV(R_{n-1}) \cup \{\vec{\pi_2}\}$$

$$FV(\vec{e_2}) \subseteq FV(R_{n-2}) \cup \{\vec{\pi_3}\}$$

$$\vdots$$

$$FV(\vec{e}_{m-1}) \subseteq FV(R_{n-m+1}) \cup \{\vec{\pi_m}\}$$

$$FV(\vec{e_m}) \subseteq FV(R_{n-m}\}$$

3. By definition, with a fresh π,
$[\![\,`(e')]\!]_{R_0,\ldots,R_n,R_{n+1}} = (\pi(R_{n+1}), \{(\pi, e'')\} :: \{(\vec{\pi_1}, \vec{e_1})\} :: \ldots :: \{(\vec{\pi_m}, \vec{e_m})\})$
4. $FV(\pi(R_{n+1})) \subseteq FV(R_{n+1}) \cup \{\pi\}$
5. Properties for $FV(e''), FV(\vec{e_1}), \ldots, FV(\vec{e}_m)$ are immediate from the I.H. □

Lemma A.2.19. *Let e be a stage-n λ^{gen}_{poly} expression with $FV(e) = \{x_1, \ldots, x_m\}$. Then,*

$$Close([\![e]\!]_{R_0,R_1,\ldots,R_n}) = Close([\![e]\!]_{R'_0,R_1,\ldots,R_n})$$

if $R_0(x_i) = R'_0(x_i)$ for any $i \in \{1..m\}$.

Proof. This is an adaptation of Lemma A.2.8 for the improved translation and $Close$. □

Lemma A.2.20. *Let e be a λ^{gen}_{poly} expression such that $e \in Val^{n+1}$. Then*

$$Close(\llbracket e \rrbracket_{\{\},R_1,\ldots,R_{n+1}}) = Close(\llbracket e \rrbracket_{R_1,\ldots,R_{n+1}})$$

Proof. This is an adaptation of Lemma A.2.10 for the improved translation and $Close$. $\square$

Lemma A.2.21. *Let e_1 be a stage-n and e_2 a stage-0 λ^{gen}_{poly} expression with no free variables. Then*

$$Close(\llbracket e_1 \rrbracket_{R_0,R_1,\ldots,R_n})[z \backslash Close(\llbracket e_2 \rrbracket_{\{\}})] = Close(\llbracket e_1[x \backslash e_2]^n \rrbracket_{R_0,R_1,\ldots,R_n})$$

where $R_0(x) = z$.

Proof. This is an adaptation of Lemma A.2.11 for the improved translation and $Close$. $\square$

Lemma A.2.22. *Let e be a stage-n λ^{gen}_{poly} expression. Then*

$$Close(\llbracket e \rrbracket_{R_0,\ldots,R_n})[r_m \backslash R_m] \longrightarrow^*_{|\beta|} Close(\llbracket e \rrbracket_{R_0[r_m \backslash R_m],\ldots,R_n[r_m \backslash R_m]})$$

Proof. This is an adaptation of Lemma A.2.14 for the improved translation and $Close$. By structural induction on e. $\square$

Proof of Theorem 5.9.7. By induction on the structure of e_1, based on the last applied reduction. We only show interesting cases.

- Case ESABS: $\mathcal{S}, \lambda x.e \longrightarrow_{n+1} \mathcal{S}', \lambda x.e'$ with the premise $\mathcal{S}, e \longrightarrow_{n+1} \mathcal{S}', e'$. Without loss of generality, assume

$$\llbracket e \rrbracket_{\{\},R_1,\ldots,R_{n+1}\ \mathsf{with}\ \{x=z\}} = (e_0, [\{\pi_1, e_1\}, \ldots, \{\pi_p, e_p\}])$$

$$\llbracket e' \rrbracket_{\{\},R_1,\ldots,R_{n+1}\ \mathsf{with}\ \{x=z\}} = (e'_0, [\{\pi'_1, e'_1\}, \ldots, \{\pi'_q, e'_q\}])$$

So,

$$Close(\llbracket e \rrbracket_{\{\},R_1,\ldots,R_{n+1}\ \mathsf{with}\ \{x=z\}}) = (\lambda \pi_p. \cdots ((\lambda \pi_1.e_0)e_1) \cdots)e_p$$

$$Close(\llbracket e' \rrbracket_{\{\},R_1,\ldots,R_{n+1}\ \mathsf{with}\ \{x=z\}}) = (\lambda \pi'_q. \cdots ((\lambda \pi'_1.e'_0)e'_1) \cdots)e'_q$$

and therefore

$$Close(\llbracket \lambda x.e \rrbracket_{\{\},R_1,\ldots,R_{n+1}}) = (\lambda \pi_p. \cdots ((\lambda \pi_1.\lambda z.e_0)e_1) \cdots)e_p$$

$$Close(\llbracket \lambda x.e' \rrbracket_{\{\},R_1,\ldots,R_{n+1}}) = (\lambda \pi'_q. \cdots ((\lambda \pi'_1.\lambda z.e'_0)e'_1) \cdots)e'_q$$

By I.H. we have

$$\llbracket \mathcal{S} \rrbracket, Close(\llbracket e \rrbracket_{\{\},R_1,\ldots,R_{n+1}\ \mathsf{with}\ \{x=z\}}) \ \longrightarrow_R \ \llbracket \mathcal{S}' \rrbracket, e'' \tag{A.7}$$

such that $e'' \longrightarrow^*_{|\beta|} Close(\llbracket e' \rrbracket_{\{\},R_1,\ldots,R_{n+1} \text{ with } \{x=z\}})$.

Recall that in staged semantics, evaluation occurs only at stage-0, or at stage-1 as a hole fill-in. Because of this, it must be that $p = n + 1$ (otherwise $\lambda x.e$ would be a stage-$n+1$ value that cannot take a step of evaluation) and there are only two possibilities:

1. A staged-0 reduction happens as part of e, meaning the premise of judgment (A.7) above is $\llbracket \mathcal{S} \rrbracket, e_p \ \longrightarrow_R \ \llbracket \mathcal{S}' \rrbracket, \overline{e}_p$.
   This makes e'' equal to $(\lambda \pi_p. \cdots ((\lambda \pi_1.e_0)e_1) \cdots)\overline{e}_p$, giving

   $$(\lambda \pi_p. \cdots ((\lambda \pi_1.e_0)e_1) \cdots)\overline{e}_p \longrightarrow^*_{|\beta|} ((\lambda \pi'_q. \cdots ((\lambda \pi'_1.e'_0)e'_1) \cdots)e'_q)$$

   Let $C[\cdot]$ be the context $(\lambda \pi_p. \cdots ((\lambda \pi_1.[\,])e_1) \cdots)\overline{e}_p$, and $C'[\cdot]$ be the context $(\lambda \pi'_q. \cdots ((\lambda \pi'_1.[\,])e'_1) \cdots)e'_q$. Then the two terms above are, respectively, $C[e_0]$ and $C'[e'_0]$; and $C[e_0] \longrightarrow^*_{|\beta|} C'[e'_0]$. The reductions included can be outside of e_0 in the context $C[\cdot]$, or directly inside e_0:
   In the former case, the context would change and become, say, $C_1[\cdot]$, and some substitutions[1] may be performed on e_0. Let us represent the effect of these substitutions as S. The term we obtain is then $C_1[Se_0]$.
   In the latter case, the context would have no change at all, but only e_0 would reduce to another term, say, $\overline{e}_0$. So the term we finally obtain is $C_1[S\overline{e}_0]$, giving $C_1[S\overline{e}_0] = C'[e'_0]$. So, $C_1[\cdot] = C'[\cdot]$ and $S\overline{e}_0 = e'_0$.
   When it comes to $Close(\llbracket \lambda x.e \rrbracket_{\{\},R_1,\ldots,R_{n+1}})$, because of the premise, we have

   $$\llbracket \mathcal{S} \rrbracket, Close(\llbracket \lambda x.e \rrbracket_{\{\},R_1,\ldots,R_{n+1}}) \longrightarrow_R \llbracket \mathcal{S}' \rrbracket, C[K[e_0]]$$

   where $K[\cdot]$ is the context $\lambda z.[\,]$. Also,

   $$Close(\llbracket \lambda x.e' \rrbracket_{\{\},R_1,\ldots,R_{n+1}}) = C'[K[e'_0]]$$

   Applying the same safe reductions for $C[\cdot]$, and e_0 above, we obtain $C[K[e_0]] \longrightarrow^*_{|\beta|} C_1[S(K[\overline{e_0}])]$. Note that the context $K[\cdot]$ binds the fresh variable z, but this variable does not exist free in $C[\cdot]$. Hence, the substitution S does not contain it, and we can safely say that $S(K[\overline{e_0}]) = K[S\overline{e_0}]$. Using the equalities above, we obtain $C_1[K[S\overline{e_0}]] = C'[K[e'_0]]$, which means that $C[K[e_0]] \longrightarrow^*_{|\beta|} Close(\llbracket \lambda x.e' \rrbracket_{\{\},R_1,\ldots,R_{n+1}})$.
2. No stage-0 evaluation occurs, but a stage-1 hole gets filled in. This means e_p is a value and $\mathcal{S} = \mathcal{S}'$, because filling in a hole does not alter the store. So, by

[1] This would be the case, for instance, of expanding a function application or a let-expression.

TRAPP, we have

$$
\begin{aligned}
&[\![\mathcal{S}]\!], Close([\![e]\!]_{\{\},R_1,\ldots,R_{n+1}\text{ with }\{x=z\}}) \\
&\qquad \longrightarrow_R \quad [\![\mathcal{S}]\!], ((\lambda\pi_{p-1}.\cdots((\lambda\pi_1.e_0)e_1)\cdots)e_{p-1})[\pi_p\backslash e_p]
\end{aligned}
$$

and

$$
\begin{aligned}
&[\![\mathcal{S}]\!], Close([\![\lambda x.e]\!]_{\{\},R_1,\ldots,R_{n+1}}) \\
&\qquad \longrightarrow_R \quad [\![\mathcal{S}]\!], ((\lambda\pi_{p-1}.\cdots((\lambda\pi_1.\lambda z.e_0)e_1)\cdots)e_{p-1})[\pi_p\backslash e_p]
\end{aligned}
$$

Note that by I.H.

$$((\lambda\pi_{p-1}.\cdots((\lambda\pi_1.e_0)e_1)\cdots)e_{p-1})[\pi_p\backslash e_p] \longrightarrow^*_{|\beta|} (\lambda\pi'_q.\cdots((\lambda\pi'_1.e'_0)e'_1)\cdots)e'_q$$

We now need to show that

$$((\lambda\pi_{p-1}.\cdots((\lambda\pi_1.\lambda z.e_0)e_1)\cdots)e_{p-1})[\pi_p\backslash e_p] \longrightarrow^*_{|\beta|} (\lambda\pi'_q.\cdots((\lambda\pi'_1.\lambda z.e'_0)e'_1)\cdots)e'_q$$

which can be done by reasoning about the contexts the same way we did above for the first case.

- Cases ESSYM, ESFIX, ESAPP(1), ESAPP(2), ESLET(1), ESLET(2), ESRUN(1), ESLIFT(1), ESREF(1), ESDEREF(1), ESASGN(1), and ESASGN(2) require using the I.H. the same way as in the ESABS case.

- Case ESAPP(3): $\mathcal{S}, (\lambda x.e_1)e_2 \longrightarrow_0 \mathcal{S}, e_1[x\backslash e_2]^0$ with the premise $e_2 \in Val^0$. Note that

$$Close([\![(\lambda x.e_1)e_2]\!]_{\{\}}) = (\lambda z.e_0)([\![e'_0]\!])$$

where $[\![e_1]\!]_{\{x=z\}} = (e_0, \texttt{nil})$ and $[\![e_2]\!]_{\{\}} = (e'_0, \texttt{nil})$. Because $e_2 \in Val^0$, we have $e'_0 \in RVal$. Hence, $SEF(e'_0)$. Then, at the record semantics side we have

$$[\![\mathcal{S}]\!], (\lambda z.e_0)(e'_0) \ \longrightarrow_R \ [\![\mathcal{S}]\!], e_0[z\backslash e'_0]$$

Note that $e_0[z\backslash e'_0] = Close([\![e_1]\!]_{\{x=z\}})[z\backslash Close([\![e_2]\!]_{\{\}})]$, which is equal to $Close([\![e_1[x\backslash e_2]^0]\!]_{\{x=z\}})$ by A.2.21, and $Close([\![e_1[x\backslash e_2]^0]\!]_{\{x=z\}}) = Close([\![e_1[x\backslash e_2]^0]\!]_{\{\}})$ by Lemma A.2.19.

- Case ESAPP(4): $\mathcal{S}, (\mathsf{fix}\, f(x).\, e_1)e_2 \longrightarrow_0 \mathcal{S}, e_1[f\backslash \mathsf{fix}\, f(x).\, e_2]^0[x\backslash e_2]^0$ with the premise $e_2 \in Val^0$. This is a case that is very similar to ESAPP(3) above.

- Case ESLET(3): $\mathcal{S}, \mathsf{let}\; x = e_1 \;\mathsf{in}\; e_2 \longrightarrow_0 \mathcal{S}, e_2[x\backslash e_1]^0$ with the premise $e_1 \in Val^0$. This is a case that is very similar to ESAPP(3).

- Case ESBOX: $\mathcal{S}, \langle e \rangle \longrightarrow_n \mathcal{S}', \langle e' \rangle$ with the premise $\mathcal{S}, e \longrightarrow_{n+1} \mathcal{S}', e'$. Without loss of generality, assume

$$[\![e]\!]_{\{\},R_1,\ldots,R_n,r} = (e_0, [\{\pi_1, e_1\}, \ldots, \{\pi_p, e_p\}])$$

$$[\![e']\!]_{\{\},R_1,\ldots,R_n,r} = (e'_0, [\{\pi'_1, e'_1\}, \ldots, \{\pi'_q, e'_q\}])$$

So,

$$Close([\![e]\!]_{\{\},R_1,\ldots,R_n,r}) = (\lambda\pi_p.\cdots((\lambda\pi_1.e_0)e_1)\cdots)e_p$$

$$Close([\![e']\!]_{\{\},R_1,\ldots,R_n,r}) = (\lambda\pi'_q.\cdots((\lambda\pi'_1.e'_0)e'_1)\cdots)e'_q$$

Hence,

$$Close([\![\langle e \rangle]\!]_{\{\},R_1,\ldots,R_n}) = (\lambda\pi_p.\cdots((\lambda\pi_1.\lambda r.e_0)e_1)\cdots)e_p$$

$$Close([\![\langle e' \rangle]\!]_{\{\},R_1,\ldots,R_n}) = (\lambda\pi'_q.\cdots((\lambda\pi'_1.\lambda r.e'_0)e'_1)\cdots)e'_q$$

By I.H. we have

$$[\![\mathcal{S}]\!], Close([\![e]\!]_{\{\},R_1,\ldots,R_n,r}) \;\longrightarrow_R\; [\![\mathcal{S}']\!], e''$$

such that $e'' \longrightarrow^*_{|\beta|} Close([\![e']\!]_{\{\},R_1,\ldots,R_n,r})$. And the rest of the proof for this case goes in the same style of the ESABS case using the I.H.

- Case ESUBOX(1): $\mathcal{S}, \grave{}(e) \longrightarrow_{n+1} \mathcal{S}', \grave{}(e')$ with the premise $\mathcal{S}, e \longrightarrow_n \mathcal{S}', e'$. Without loss of generality, assume

$$[\![e]\!]_{\{\},R_1,\ldots,R_n} = (e_0, [\{\pi_1, e_1\}, \ldots, \{\pi_p, e_p\}])$$

$$[\![e']\!]_{\{\},R_1,\ldots,R_n} = (e'_0, [\{\pi'_1, e'_1\}, \ldots, \{\pi'_q, e'_q\}])$$

So,

$$Close([\![e]\!]_{\{\},R_1,\ldots,R_n}) = (\lambda\pi_p.\cdots((\lambda\pi_1.e_0)e_1)\cdots)e_p$$

$$Close([\![e']\!]_{\{\},R_1,\ldots,R_n}) = (\lambda\pi'_q.\cdots((\lambda\pi'_1.e'_0)e'_1)\cdots)e'_q$$

Hence,

$$Close([\![\grave{}(e)]\!]_{\{\},R_1,\ldots,R_n,R_{n+1}}) = (\lambda\pi_p.\cdots((\lambda\pi_1.((\lambda\pi_0.\pi_0(R_{n+1}))e_0))e_1)\cdots)e_p$$

$$Close([\![\grave{}(e')]\!]_{\{\},R_1,\ldots,R_n,R_{n+1}}) = (\lambda\pi'_q.\cdots((\lambda\pi'_1.((\lambda\pi'_0.\pi'_0(R_{n+1}))e'_0))e'_1)\cdots)e'_q$$

By I.H. we have

$$[\![\mathcal{S}]\!], Close([\![e]\!]_{\{\},R_1,\ldots,R_n}) \;\longrightarrow_R\; [\![\mathcal{S}']\!], e''$$

such that $e'' \longrightarrow^*_{|\beta|} Close([\![e']\!]_{\{\},R_1,\ldots,R_n})$. And the rest of the proof for this case goes in the same style of the the ESABS case using the I.H.

- Case ESUBOX(2): $\mathcal{S}, \text{`}(\langle e \rangle) \longrightarrow_1 \mathcal{S}, e$ with the premise $e \in Val^1$. Because of the premise, we have $[\![e]\!]_{\{\},r} = (e_0, \mathtt{nil})$.
  Therefore, $[\![\langle e \rangle]\!]_{\{\}} = (\lambda r.e_0, \mathtt{nil})$ and $Close([\![\text{`}(\langle e \rangle)]\!]_{\{\},R_1}) = (\lambda \pi_0.\pi_0 R_1)(\lambda r.e_0)$. By ERAPP, we have

$$[\![\mathcal{S}]\!], (\lambda \pi_0.\pi_0 R_1)(\lambda r.e_0) \longrightarrow_R [\![\mathcal{S}]\!], (\lambda r.e_0) R_1$$

  Note that $(\lambda r.e_0) R_1 \longrightarrow_{|\beta|} e_0[r \backslash R_1]$. Using the fact that $e_0 = Close([\![e]\!]_{\{\},r})$, we have $Close([\![e]\!]_{\{\},r})[r \backslash R_1] \longrightarrow^*_{|\beta|} Close([\![e]\!]_{\{\},R_1})$ by Lemma A.2.22.

- Case ESRUN(2): $\mathcal{S}, \mathsf{run}(\langle e \rangle) \longrightarrow_0 \mathcal{S}, e$ with the premise $e \in Val^1$. Because of the premise, we have $[\![e]\!]_{\{\},r} = (e_0, \mathtt{nil})$.
  Therefore, $[\![\langle e \rangle]\!]_{\{\}} = (\lambda r.e_0, \mathtt{nil})$ and $Close([\![\mathsf{run}(\langle e \rangle)]\!]_{\{\}}) = (\lambda r.e_0)\{\}$. By ERAPP, we have

$$[\![\mathcal{S}]\!], (\lambda r.e_0)\{\} \longrightarrow_R [\![\mathcal{S}]\!], e_0[r \backslash \{\}]$$

  Using the fact that $e_0 = Close([\![e]\!]_{\{\},r})$, we have $Close([\![e]\!]_{\{\},r})[r \backslash \{\}] \longrightarrow^*_{|\beta|} Close([\![e]\!]_{\{\},\{\}})$ by Lemma A.2.22. And finally $Close([\![e]\!]_{\{\},\{\}}) = Close([\![e]\!]_{\{\}})$ by Lemma A.2.20.

- Case ESLIFT(2): $\mathcal{S}, \mathsf{lift}(e) \longrightarrow_0 \mathcal{S}, \langle e \rangle$ with the premise $e \in Val^0$. Because of the premise, we have $[\![e]\!]_{\{\}} = (e_0, \mathtt{nil})$. Therefore, $Close([\![\mathsf{lift}(e)]\!]_{\{\}}) = \mathsf{let}\, \pi = e_0 \,\mathsf{in}\, \lambda r.\pi$. Since $e \in Val^0$, we have $e_0 \in RVal$. By ERLET we have

$$[\![\mathcal{S}]\!], (\mathsf{let}\, \pi = e_0 \,\mathsf{in}\, \lambda r.\pi) \longrightarrow_R [\![\mathcal{S}]\!], \lambda r.e_0$$

  Using the fact that $e_0 = Close([\![e]\!]_{\{\}})$, we have $\lambda r.e_0 = Close([\![e]\!]_{\{\}})$. By Lemma A.2.19, $Close([\![e]\!]_{\{\}}) = Close([\![e]\!]_r)$. And by Lemma A.2.20, we get $Close([\![e]\!]_r) = Close([\![e]\!]_{\{\},r}) = (e_0, \mathtt{nil})$. Hence, $Close([\![\langle e \rangle]\!]_{\{\}}) = \lambda r.e_0$.

- Case ESREF(2): $\mathcal{S}, \mathsf{ref}\ e \longrightarrow_0 \mathcal{S} \triangleleft\!\!+ \{\ell : e\}, \ell$ with the premise $e \in Val^0$ and $\ell \notin dom(\mathcal{S})$.

  Note that, because $e \in Val^0$, we have $[\![e]\!]_{\{\}} = (e_0, \mathtt{nil})$ and $e_0 \in RVal$. Therefore, $Close([\![\mathsf{ref}\ e]\!]_{\{\}}) = \mathsf{ref}\ e_0$, and by ERREF

$$[\![\mathcal{S}]\!], \mathsf{ref}\ e_0 \longrightarrow_R [\![\mathcal{S}]\!] \triangleleft\!\!+ \{\ell : e_0\}, \ell$$

  Trivially, $\ell \longrightarrow^*_{|\beta|} \ell$, and $[\![\mathcal{S} \triangleleft\!\!+ \{\ell : e\}]\!] = [\![\mathcal{S}]\!] \triangleleft\!\!+ \{\ell : e_0\}$ because $Close([\![e]\!]_{\{\}}) = e_0$.

- Case ESDEREF(2): $\mathcal{S}, !\ell \longrightarrow_0 \mathcal{S}, v$ with the premise $\mathcal{S}(\ell) = v$.

  Note that $Close([\![!\ell]\!]_{\{\}}) = !\ell$ and $Close([\![v]\!]_{\{\}}) = (v_0, \mathtt{nil})$ and $[\![\mathcal{S}]\!](\ell) = v_0$. Hence, by ERDEREF

$$[\![\mathcal{S}]\!], !\ell \longrightarrow_R [\![\mathcal{S}]\!], v_0$$

- Case ESASGN(3): $\mathcal{S}, \ell{:=}e_2 \longrightarrow_0 \mathcal{S} \Leftarrow \{\ell : e_2\}, e_2$ with the premise $e_2 \in Val^0$.

  Note that, because $e_2 \in Val^0$, we have $[\![e_2]\!]_{\{\}} = (e_0, \texttt{nil})$ and $e_0 \in RVal$. Therefore, $Close([\![\ell{:=}e_2]\!]_{\{\}}) = \ell{:=}e_0$, and by ERREF

  $$[\![\mathcal{S}]\!], \ell{:=}e_0 \longrightarrow_R [\![\mathcal{S}]\!] \Leftarrow \{\ell : e_0\}, e_0$$

  Recall that $e_0 = Close([\![e_2]\!]_{\{\}})$. □

Proof of Theorem 5.9.10. By structural induction on e_1. This proof is similar to Theorem 5.6.4 with additional use of the fact that the reduction does not alter the unmatched holes inside expressions if the stage is greater than 0, and that there are no unmatched holes if the stage is 0. □

Proof of Theorem 5.9.13. By induction on the structure of e. The proof frequently uses Lemma A.2.1 based on the fact obtained from Lemma A.2.18. □

Bibliography

[AC96] Martín Abadi and Luca Cardelli. A theory of primitive objects: untyped and first-order systems. *Inf. Comput.*, 125(2):78–102, 1996.

[ACK03] Giuseppe Attardi, Antonio Cisternino, and Andrew Kennedy. Codebricks: code fragments as building blocks. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 66–74. ACM Press, 2003.

[Adv] Advanced Programming for the Java 2 Platform: Ch. 8; Performance Features and Tools. `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/perf2.html`.

[AJKC05] Baris Aktemur, Joel Jones, Sam Kamin, and Lars Clausen. Optimizing marshalling by run-time program generation. In Robert Glück and Michael R. Lowry, editors, *GPCE '05: Proceedings of the 4th international conference on Generative programming and component engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2005.

[AK05] Baris Aktemur and Sam Kamin. Mumbo: A rule-based implementation of a run-time program generation language. In *Proc. of the 6th Intl. Workshop on Rule-Based Programming*, April 2005. Nara, Japan.

[AK09] Baris Aktemur and Sam Kamin. Writing customizable libraries - a comparative study. In *The 24th Annual ACM Symposium on Applied Computing (SAC)*, Honolulu, HI, USA, 2009.

[Akt05] Baris Aktemur. A rule-based model of a run-time program generation system. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 2005.

[Asp] Aspectj web site. `http://www.aspectj.org`.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Baw99] Alan Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-based Program Manipulation*, pages 4–12, 1999.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[Cha02] Craig Chambers. Staged compilation. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 1–8, New York, NY, USA, 2002. ACM.

[Cla04] Lars Clausen. *Optimizations In Distributed Run-time Compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[CLM04] Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur. A tour of tempo: a program specializer for the c language. *Sci. Comput. Program.*, 52(1-3):341–370, 2004.

[CMT00] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 25–36, London, UK, 2000. Springer-Verlag.

[CMT04] C Calcagno, E Moggi, and W Taha. Ml-like inference for classifiers. *Programming Languages and Systems*, 2986:79–93, 2004.

[COST04] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72, 2004.

[CX03] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 275–286, New York, NY, USA, 2003. ACM.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 184, Washington, DC, USA, 1996. IEEE Computer Society.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–270, New York, NY, USA, 1996. ACM.

[EHK96] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: a language for high-level, efficient, and machine-independent dynamic code generation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 131–144. ACM Press, 1996.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1. Elsevier, 1995.

[FCL06] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 275–284, New York, NY, USA, 2006. ACM.

[Fre97] Alexandre Frey. Satisfying subtype inequalities in polynomial space. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 265–277, London, UK, 1997. Springer-Verlag.

[Fut99] Yoshihiko Futamura. Partial evaluation of computation process—anapproach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, 1999. This is an updated and revised version of "Partial Evaluation of Computation Process—an Approach to a Compiler-Compiler", originally published in "Systems Computers Controls", Volume 2, Number 5, 1971, pages 45–50.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université de Paris VII, 1972.

[GJ95] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *PLILPS '95: Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 259–278, London, UK, 1995. Springer-Verlag.

[GJ97] Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp Symb. Comput.*, 10(2):113–158, 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Definition*. Addison-Wesley, 1996.

[GMP+00] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, 1994.

[HZS05] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with safegen. In R. Glueck and M. Lowry, editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326, Tallinn, Estonia, September 2005. Springer.

[HZS07a] Shan Shan Huang, David Zook, and Yannis Smaragdakis. cj: Enhancing java with safe type conditions. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 185–198. ACM Press, 2007.

[HZS07b] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 399–424. Springer-Verlag, 2007.

[IB99] Andrew Ireland and Alan Bundy. Automatic verification of functions with accumulating parameters. *J. Funct. Program.*, 9(2):225–245, 1999.

[IBM] IBM-JVM. `http://www-106.ibm.com/developerworks/java/jdk/`.

[Java] Java 1.4.2 API Documentation. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[Javb] Java hotspot virtual machine. `http://java.sun.com/javase/technologies/hotspot/`.

[Javc] Java Object Serialization Specification. `http://java.sun.com/j2se/1.4.2/docs/guide/serialization/spec/serialTOC.html`.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Jim96] Trevor Jim. What are principal typings and what are they good for? In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–53, St. Petersburg Beach, Florida, United States, 1996.

[JSS85] Neil D. Jones, Peter Sestoft, and Harald Sondergaard. An experiment in partial evaluation: the generation of a compiler generator. In *Proc. of the first international conference on Rewriting techniques and applications*, pages 124–140, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[Kaf] Kaffe JVM. `http://www.kaffe.org`.

[KAK06] Sam Kamin, Baris Aktemur, and Michael Katelman. Staging static analyses for program generation. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 1–10, New York, NY, USA, 2006. ACM.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 311–320, New York, NY, USA, 2008. ACM.

[Kam03] Sam Kamin. Routine run-time code generation. *SIGPLAN Not.*, 38(12):44–56, 2003.

[Kam04] Sam Kamin. Program generation considered easy. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–79. ACM Press, 2004.

[KAM05] Sam Kamin, Baris Aktemur, and Philip Morton. Source-level optimization of run-time program generators. In Robert Glück and Michael R. Lowry, editors, *GPCE '05: Proceedings of the 4th international conference on Generative programming and component engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.

[Kat06] Michael Katelman. Staged static analyses and run-time program generation. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 2006.

[KCC00a] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components-1: Source-level components. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 49–64. Springer-Verlag, 2000.

[KCC00b] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components-2: Binary-level components. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 28–50. Springer-Verlag, 2000.

[KCJ03] Sam Kamin, Lars Clausen, and Ava Jarvis. Jumbo: run-time code generation for java and its applications. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 48–56. IEEE Computer Society, 2003.

[KGS94] R. Kramer, R. Gupta, and M. L. Soffa. The combining dag: A technique for parallel data flow analysis. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):805–813, 1994.

[KKcS08] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Closing the stage: from staged code to typed closures. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 147–157, New York, NY, USA, 2008. ACM.

[KYC06] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 257–268. ACM Press, 2006.

[Lea06] Christopher League. Metaocaml server pages: web publishing as staged computation. *Sci. Comput. Program.*, 62(1):66–84, 2006.

[LR94] Yong-Fong Lee and Barbara G. Ryder. Effectively exploiting parallelism in data flow analysis. *J. Supercomput.*, 8(3):233–262, 1994.

[LRF95] Yong-Fong Lee, Barbara G. Ryder, and Marc E. Fiuczynski. Region analysis: A parallel elimination method for data flow analysis. *IEEE Trans. Softw. Eng.*, 21(11):913–926, 1995.

[LRM91] Yong-Fong Lee, Barbara G. Ryder, and Thomas J. Marlowe. Experiences with a parallel algorithm for data flow analysis. *J. Supercomput.*, 5(2-3):163–188, 1991.

[Mor05] Philip Morton. Analyses and rewrites for optimizing jumbo. Master's thesis, University of Illinois at Urbana-Champaign, 2005.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, New York, NY, USA, 1990. ACM.

[MTBS99] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized metaml: Simpler, and more expressive. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[MvNV+01] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman. Efficient Java RMI for Parallel Programming. *ACM Trans. Program. Lang. Syst.*, 23(6):747–775, 2001.

[Nan02] Aleksandar Nanevski. Meta-programming with names and necessity. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 206–217, New York, NY, USA, 2002. ACM.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, New York, NY, USA, 1992.

[NPH99] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Proc. of the ACM 1999 Java Grande*, pages 152–159, New York, NY, USA, 1999. ACM Press.

[NR04] Gregory Neverov and Paul Roe. Metaphor: A multi-stage, object-oriented programming language. In G. Karsai and E. Visser, editors, *Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *Lecture Notes in Computer Science*, Vancouver, Canada, October 2004. Springer.

[OMY01] Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. Dynjava: Type safe dynamic code generation in java. In *The 3rd JSSST Workshop on Programming and Programming Languages (PPL2001)*, March 2001.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[Pal95] Jens Palsberg. Efficient inference of object types. *Inf. Comput.*, 123(2):198–209, 1995.

[PHEK99] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pot00a] François Pottier. `Wallace`: an efficient implementation of type inference with subtyping, February 2000.

[Pot00b] François Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.

[PWO97] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9(1):49–67, 1997.

[PZ02] Jens Palsberg and Tian Zhao. Efficient type inference for record concatenation and subtyping. *lics*, 00:125, 2002.

[Reh98] Jacop Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, 1998.

[Rém94] Didier Rémy. Type inference for records in natural extension of ml. *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 67–95, 1994.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag.

[Rhi05] Morten Rhiger. First-class open and closed code fragments. In *Proceedings of the Sixth Symposium on Trends in Functional Programming*, pages 127–144, Tallinn, Estonia, 2005.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, New York, NY, USA, 1995. ACM.

[RKM06] Atanas Rountev, Scott Kagan, and Thomas Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction*, LNCS 3923, pages 2–16, 2006.

[SGM+03] Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for template-based run-time code generation. *Journal of Functional Programming*, 13(3):677–708, 2003.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[Sto77] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[Tah03] Walid Taha. A gentle introduction to multi-stage programming. *Domain-Specific Program Generation*, 3016:30–50, 2003.

[TCLP] Walid Taha, Cristiano Calcagno, Xavier Leroy, and Ed Pizzi. Metaocaml. http://www.metaocaml.org/.

[TN03] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–37, New York, NY, USA, 2003. ACM.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, NY, USA, 1997. ACM.

[TS00] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[vNMW+05] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger F. H. Hofman, Ceriel J. H. Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: A Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005.

[VP03] Ronald Veldema and Michael Philippsen. Compiler Optimized Remote Method Invocation. In *IEEE International Conference on Cluster Computing (CLUSTER'03)*, page 127, December 2003.

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, 1991.

[Wel94] J. B. Wells. Typability and type-checking in the second-order lambda-calculus are equivalent and undecidable. In *Logic in Computer Science*, pages 176–185, 1994.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, 1995.

[YI06] Yosihiro Yuse and Atsushi Igarashi. A modal type system for multi-level generating extensions with persistent code. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 201–212, New York, NY, USA, 2006. ACM.

[ZHS04] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating aspectj programs with meta-aspectj. In G. Karsai and E. Visser, editors, *Proc. of the Third Intl. Conf. on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *Lecture Notes in Computer Science*, pages 1–18, Vancouver, Canada, October 2004. Springer.

Author's Biography

Tankut Barış Aktemur was born and raised in the city of Samsun, Turkey, at the southern coast of the Black Sea, where he stayed until he moved to Ankara to study computer science at Bilkent University. He then received a master's degree from the University of Illinois at Urbana-Champaign on the way to his Ph.D. Barış likes blue skies with scattered clouds, snow-topped mountains, honey bunches of oats with almonds, sunset at the Aegean sea, migrating geese, Coltrane's version of "My Favorite Things", and the cinnamon-carrot-walnut cake his mother bakes.