THE DESIGN OF INTERMEDIATE LANGUAGES IN OPTIMIZING COMPILERS

by

LUKE VAN WYCK MAURER

A DISSERTATION

Presented to the Department of Computer & Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

March 2018

DISSERTATION APPROVAL PAGE

Student: Luke Van Wyck Maurer

Title: The Design of Intermediate Languages in Optimizing Compilers

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer & Information Science by:

Zena M. Ariola	Chair
Michal Young	Core Member
Boyana Norris	Core Member
Mark Lonergan	Institutional Representative

and

Sara D. Hodges	Interim Vice Provost and Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded March 2018

DISSERTATION ABSTRACT

Luke Van Wyck Maurer

Doctor of Philosophy

Department of Computer and Information Science

March 2018

Title: The Design of Intermediate Languages in Optimizing Compilers

Every compiler passes code through several stages, each a sort of mini-compiler of its own. Thus each stage may deal with the code in a different representation, which may have little to do with the source or target language. We can describe these in-memory representations as languages in their own right, which we call *intermediate languages*.

Each intermediate language is designed to accomodate the stage of compilation that handles it. Those toward the end of the compilation pipeline, for instance, tend to have features expressing low-level details of computation. A subtler case is that of the optimization stage, whose role is to transform the program so that it runs faster, uses less memory, and so forth. The optimizer faces tradeoffs: The language should provide enough information to guide optimization algorithms, but all of this information must be kept up to date as the program is transformed. Also, establishing invariants in the language can be helpful both in implementing algorithms and in debugging the implementation, but each invariant may complicate desirable transformations or rule them out altogether. Finally, a language where the invariants are obviously correct may have a form too awkward or otherwise unsuited to the compiler's needs.

Given the properties and invariants that we would like the language to provide, we can approach the design task in a way that gives these features without necessarily sacrificing implementability. Namely, begin with a formal language that makes the desired properties obvious, then translate it to one more suitable for implementation. We can even translate theorems about valid transformations in the formal language to derive correct algorithms in the implementation language.

This dissertation explores the connections between different intermediate languages and how they can be interderived, then demonstrates how translation lead to an improvement to the Glasgow Haskell Compiler opimization engine.

This dissertation includes previously published coauthored material.

CURRICULUM VITAE

NAME OF AUTHOR: Luke Van Wyck Maurer

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR

Carleton College, Northfield, MN

DEGREES AWARDED:

Master of Science, Computer & Information Science, 2012, University of Oregon

Bachelor of Arts, Mathematics and Computer Science, 2006, Carleton College

AREAS OF SPECIAL INTEREST:

Programming Languages
Compilation
Formal Methods
Proof Assistants

PROFESSIONAL EXPERIENCE:

Junior Software Developer, SingleMind Consulting, September 2006–October 2008

Software Developer, Emberex Custom Software, December 2008–June 2011

Intern, Microsoft Research, summer 2015

GRANTS, AWARDS AND HONORS:

Distinguished Paper, 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)

PUBLICATIONS:

Downen, P., Maurer, L., Ariola, Z. M., & Varacca, D. (2014) Continuations, processes, and sharing. PPDP 2014.

Downen, P., Maurer, L., Ariola, Z. M., & Peyton Jones, S. (2016) Sequent calculus as a compiler intermediate language. ICFP 2016.

Maurer, L., Downen, P., Ariola, Z. M., & Peyton Jones, S. (2017) Compiling without continuations. PLDI 2017.

ACKNOWLEDGEMENTS

I thank Professors Ariola and Young for their assistance and encouragement (as well as their patience). Thanks also to INRIA and to Microsoft Research for the internships that supported much of this work, and especially Daniele Varacca of INRIA and Simon Peyton Jones of Microsoft Research for their guidance. Special thanks to Paul Downen, Philip Johnson-Freyd, and Zach Sullivan for endless discussions on the finer points of programming-language theory and also nuclear politics. This research was supported by a grant from the National Science Foundation, CCF-1423617, to Prof. Ariola.

For my parents, who taught me the value of hard work by raising me; and for Mrs. Klinge and all my other teachers at Logos Lab School, who sparked the researcher in me and gave me a taste of grad school at age eleven. (I regret that this project did not include a diorama.)

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

Introduction

The compiler is a complex beast. As a program transforms from a source code comprehensible to humans into a machine language suitable for hardware execution, it must pass through several stages, including *parsing*, *optimization*, and *code generation*. At each stage, the code may be translated from one form to another. Some of these forms, such as *abstract syntax trees* (ASTs), are closely related to the source or target languages, but many fall in between: they are *intermediate representations*, or IRs.

IRs serve several purposes. They are essential for mitigating the complexity of any minimally sophisticated compiler. Also, they make the compiler more flexible, and thus more useful, by abstracting out finer details of the source or target language, thus allowing the same compiler to target different hardware platforms or to implement several source languages.

Here we focus on another use for IRs: Since they represent the program more abstractly than either the source language or the target, they are ideal for the sort of manipulation performed by an optimizer. Optimizers must work carefully to avoid changing behavior, so a simpler IR is better: there are fewer opportunities to mess something up. However, if we can't be confident that it's *improving* performance, it's hardly an optimizer, so a richer IR is better: optimization decisions can be better informed. The obvious tension here calls for careful attention when designing an IR.

A good way to have a rich IR while controlling for its complexity is to formalize it as a mathematical calculus. This is less daunting than it may sound: there are decades of results in programming-language theory to build on. A formalized language is as well-specified as one could ask, eliminating any question of undocumented corner cases or inconsistent requirements. Of course, designing a formalized IR still requires ironing out all such wrinkles; we're not saving any of that work. But a formal semantics certifies that the work is done.

A formal specification does far more than merely define a language, however. By establishing the rules for how the language works, it allows theorems to be proved about what modifications can be made to a program without changing its validity or its behavior. In turn, these theorems specify optimization and translation passes that can be implemented with confidence. Thus the formal specification equips compiler writers to implement transforms that are correct but not obviously so. Furthermore, it is natural when writing optimizer code to run into subtle corner cases; writing out the case in the formal language often makes the correct implementation clear, easing both the task of writing the implementation and of documenting it in comments. In this way, a formal IR can help keep compiler code maintainable while implementing highly sophisticated algorithms.

One further benefit to a formal language is that it can include types and other invariants in the specification, and these invariants can be checked programmatically. This can provide an invaluable debugging tool, as running the optimizer with the internal typechecker enabled will catch most transformation errors as soon as possible—it is rare that a buggy pass produces incorrect yet well-typed code.

Just as different languages serve different purposes in the compiler, different *choices* of language bring out various aspects of computation. For example, the pure λ-calculus has no clear representation of control flow: the sequence of actions to take is entirely implicit in the syntax. This is a natural perspective for considering the mathematical content of the program, but it is ill-suited for reasoning about sequential execution on hardware. A language in continuation-passing style, in contrast, is more complex but offers constructs that directly correspond to labeled sections of code in the running program.

Translating programs is not difficult—certainly it tends to be simpler than a typical analysis for an optimization pass. Much more difficult is proving that a translation is correct, i.e. that it preserves the meaning of a program. Once we have done so, however, we can now "translate" theorems and other facts from one language to another. In this way, we can reason about code using the best language for proving the theorems we want, then implement an optimizer that uses the most convenient language for the actual program.

1.1 Outline

Chapter II introduces several intermediate languages and discuss their origins, their purposes, and the relationships between them.

In Chapter III, we consider GHC's Core language and an extension that adds jumps and labeled blocks for explicit representation of control flow. This gives code-motion techniques more precision and flexibility, providing more opportunities to discover interactions at compile time: moving a case analysis to where a value is produced, for instance, may resolve the decision at compile time, allowing other branches to be pruned and potentially even avoiding the allocation of the produced value.

Chapter IV dives deeper into the mechanism of laziness underpinning the GHC run-time system. By drawing a connection to the π-calculus, a language for describing intercommunicating processes running in parallel, we derive an IL that describes the in-place update used by GHC-compiled code to cache results of suspended computations. We hope that a well-tailored language that includes this effect may allow updates to participate in code motion as well.

1.2 Co-Authored Material

This dissertation owes much of its content to co-authored papers already published.

- Chapter III was a collaboration with Paul Downen (University of Oregon), Zena M. Ariola (UO), and Simon Peyton Jones (Microsoft Research).
- Chapter IV was a collaboration with Paul Downen, Zena M. Ariola, and Daniele Varacca (Université Paris Diderot).

CHAPTER II

Intermediate Languages

Optimizers are necessarily heuristic and conservative—they need to gather data to make informed decisions, both in the hope of improving the code and to be certain not to introduce errors. Thus, an informative IR is a must. Each property the IR tracks has a cost, however, both in the resources for calculating and storing the property and in the need to maintain it whenever the code is altered. The design space is large, therefore, and the criteria for a good IR depend on the form of the source language (or previous IR), the requirements of the target architecture (or next IR), and the set of algorithms to be performed by the particular compilation stage. Generally speaking, a good IR is

1. straightforward to build from the incoming code, with a clear mapping for each language construct;
2. sufficiently informative to support optimization, so that common analyses need only be implemented once;
3. sufficiently flexible to allow changes to be made while preserving correctness and consistency; and
4. straightforward to translate into the next representation.

To study the breadth of possibilities, authors and researchers look at IRs not as mere data structures but as programming languages unto themselves, with well-defined semantics independent of a particular implementation. Such an *intermediate language* (IL) crystallizes the design of an IR, giving it a precise characterization removed from the intricate details of the implementation, and allowing its properties to be stated, proved, and compared to those of other languages. This paper will look at a few of the most important kinds of ILs for optimization, studying their features and differences, and finally proposing one for use in the Glasgow Haskell Compiler (GHC).

The plan is as follows: Compilers can be broadly categorized by the families of languages they implement. As such, we will look at ILs for *imperative* languages (Section 2.1) and *functional* languages (Section 2.2) separately. In Section 2.3, we will look at the specific challenges facing GHC, stemming from the particular features of the lazy functional language Haskell. Then, in Section 2.4, we introduce our own IL, which was implemented as an experimental patch to GHC.

2.1 Languages for Imperative Optimizations

Three-Address Code

A three-address code is a language comprised of linear sequences of instructions of fixed size, resembling assembly code. A typical instruction might be

$$A \leftarrow B + C,$$

indicating a destructive assignment of $B + C$ to the variable A. Depending on the particular language, A, B, and C might be abstract variables, or they might be well-defined registers or memory locations on a target architecture. In the latter case, we have abstracted away only the precise syntax of assembly language, but no other details of the target hardware, making this an ideal form for *peephole optimization* (Davidson & Fraser, 1980). This is the lowest-level form of optimization, operating at the level of individual instructions, looking only at a small window (a "peephole") at a time to find opportunities to combine instructions and save CPU cycles.

Typically, a three-address code expresses control through *labels* and *jumps*. Each instruction may carry a label identifying it as a possible target for a jump. A jump can be either *conditional*, occurring only if some condition is met, or *unconditional.* Thus a routine in three-address code to sum the members of an array a of length n might be:

```
  $i \leftarrow 0$
  $s \leftarrow 0$
$loop$:
  $c \leftarrow i - n$
  **ifge** $c$ **then** $done$
  $t \leftarrow a @ i$
  $s \leftarrow s + t$
  $i \leftarrow i + 1$
  **jump** $loop$
$done$:
  **return** $s$
```

Here **ifge** c *done* is a conditional jump that executes when $c \leq 0$, and **jump** *loop* is an unconditional jump. Note that any complex expressions have been broken down—the programmer probably wrote `s += a[i]`, but this three-address code requires the array access and the addition to be separate instructions.

Control-Flow Graphs

Three-address codes are effective for expressing programs, but a simple list of instructions is unwieldy for performing analyses and manipulation. Therefore it is typical to construct a *control-flow graph*, or CFG, to represent the control flow as edges connecting fragments of the program.

Note that CFGs do not themselves comprise a language; the CFG is an *in-memory* representation that holds code in an underlying language, such as a three-address code. Nonetheless, it is important to the study of intermediate

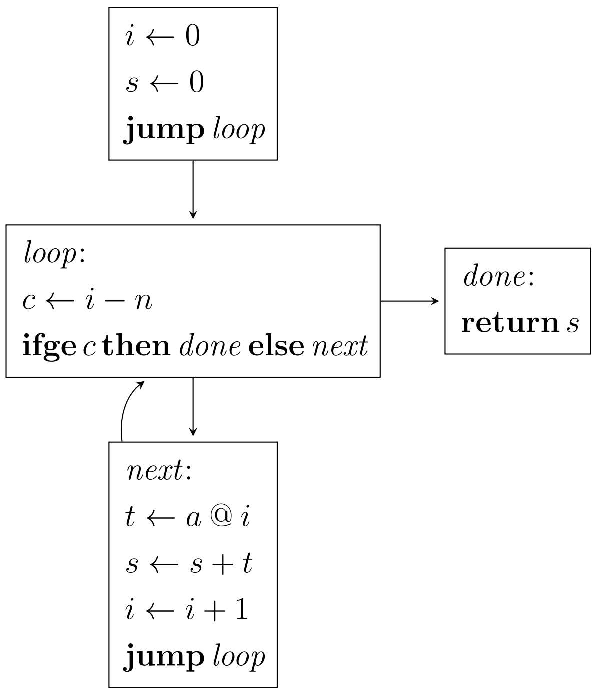


FIGURE 1. The control-flow graph for a simple array-sum program.

languages because the static single-assignment form, which we consider next, exploits the CFG that is assumed to be available.

A vertex in a CFG embodies a *basic block*, a sequence of instructions that proceeds in a straight line. No instruction in the program jumps into the middle of a basic block, and the only jump in the block can come at the end. Thus all internal jumps go from the end of one basic block to the beginning of another. The CFG then records each block as a vertex and each jump as an edge. For a given block, the blocks that may jump to it are its *predecessors* and the blocks it may jump to are its *successors*. See Fig. 1 for an example.

The CFG makes reasoning about and manipulating the control flow much easier. For instance, the basic form of *dead-code elimination* (DCE) finds instructions that will never run and deletes them. Without a CFG, one would have to scan the code for jumps to find labels that are never targeted; with a CFG, one simply checks which blocks have no predecessors.

$$\begin{array}{l} a \leftarrow\ x+y \\ b \leftarrow\ a+3 \\ c \leftarrow\ b+z \\ d \leftarrow\ a+3 \\ a \leftarrow\ d+c \\ e \leftarrow\ a+3 \\ \textbf{return}\ e \end{array} \Rightarrow \begin{array}{l} a \leftarrow\ x+y \\ b \leftarrow\ a+3 \\ c \leftarrow\ b+z \\ a \leftarrow\ b+c \\ e \leftarrow\ a+3 \\ \textbf{return}\ e \end{array}$$

FIGURE 2. An example of common-subexpression elimination.

Static Single-Assignment Form

Consider the code in Fig. 2. Clearly the computation of d is redundant; we should remove it and replace references to d with b. This is an important optimization called *common-subexpression elimination*, or CSE. But note something crucial to this analysis: the value of a did not change between the assignments to b and d. We *cannot* remove e in the same way, because "$a + 3$" is not the same value that it was when b or d computed it. Thus, starting from simple three-address code, any CSE routine must perform some analysis involving so-called *available expressions* and *reaching definitions*, walking through the code and working out all the ramifications of each assignment for other instructions (Appel, 1998a; Aho, Sethi, & Ullman, 1986).

This need is the basis of *dataflow analysis.* At its core is the question, "What are the possible values of this computation?"

The chief cause of complexity in dataflow analysis is *mutability.* If there is an intervening assignment to a variable, then two different occurrences of the variable generally don't refer to the same value, and thus we can't take an expression "at face value." In Fig. 2, the expression $a + 3$ has no consistent, well-defined value, since the value of a changes.

Traditionally, it was up to each optimization to take mutability into account so that the optimizer only makes valid changes. This added complexity to many individual algorithms. The *static single-assignment form*, or SSA form, makes dataflow obvious by eliminating mutability of variables, thus simplifying many algorithms and making new ones more feasible.

The code in Fig. 2 is not in SSA form, since there are two assignments to a. However, in this case we can observe that there are really two "versions" of a involved, and each occurrence of a refers unambiguously to one or the other. The assignments to b and d refer to the first version, and the assignment to e refers to the second. Therefore we can rename the second version to a': We have

$$\begin{array}{lll} a & \leftarrow & x + y \\ b & \leftarrow & a + 3 \\ c & \leftarrow & b + z \\ a' & \leftarrow & b + c \\ e & \leftarrow & a' + 3 \\ \multicolumn{3}{l}{\textbf{return } e} \end{array}$$

obtained the SSA form, guaranteeing that $a + 3$ has a consistent value so long as a is in scope.

Renaming suffices for only the simplest cases. Here, for any instruction, we know which "version" of a is active and thus whether to rename each occurrence of a to a'. In the presence of control flow, however, one cannot always know what the "current version" is. Thus we need a way to merge together different possible values for a given variable.

The ϕ-Node

The construct at the heart of SSA is the *ϕ-node*. A ϕ-node is an instruction of the form

$$A \leftarrow \phi(B_1, \ldots, B_n)$$

```
    $i_0 \leftarrow 0$
    $s_0 \leftarrow 0$
$loop$:
    $i \leftarrow \phi(i_0,\ i')$
    $s \leftarrow \phi(s_0,\ s')$
    $c \leftarrow i - n$
    **ifge** $c$ **then** *done* **else** *next*
$next$:
    $t \leftarrow a\,@\,i$
    $s' \leftarrow s + t$
    $i' \leftarrow i + 1$
    **jump** *loop*
$done$:
    **return** $s$
```

FIGURE 3. A routine to sum the elements of an array, in SSA form.

appearing at the beginning of a basic block. There should be one argument for each predecessor to the block. Operationally, it is understood that A will get the value B_i if the block is entered from its ith predecessor. Thus the conditional update of a variable is modeled by the creation of a *new* variable whose value depends on the control flow. See Fig. 3 for an example, where the index i will be zero when *loop* is entered from the beginning of the program but i' when it is entered from the **jump**. Since i' is $i + 1$, this causes i to be incremented each time through the loop, as expected. The accumulator variable s works similarly.

In some ways, SSA form does for dataflow what the CFG does for control flow by making crucial properties obvious. For instance, another form of dead-code elimination concerns *dead stores*, which happen when a value is written that will never be read. This can happen when two writes are made to the same variable in succession; the first is a dead store and is wasted. Without SSA, finding dead stores requires performing a *liveness analysis* by scanning backward; with SSA, there *cannot be* two writes to the same variable, so a dead store is simply a variable that is never read. Typical implementations maintain a *def-use chain* (Cytron, Ferrante, Rosen, Wegman, & Zadeck, 1991) for each variable, listing the instructions where it is used; finding a dead store is then a simple matter of checking for an empty def-use chain.

SSA is powerful enough to make new optimizations practical as well. For instance, one of the original applications (Rosen, Wegman, & Zadeck, 1988) was a generalization of CSE called *global value numbering*: once we *can* trust the face value of an expression $a + b$ because the values of a and b can't change, it becomes practical to, say, identify $a + b$ with $b + a$.

Since its inception, SSA has become the dominant form of IL both in the literature and in compilers for imperative languages—GCC, in versions 4.0 onward, uses SSA as its high-level optimization IL (Novillo, 2003; Pop, 2006), and the LLVM framework's bytecode language is in SSA form (Lattner & Adve, 2004; "LLVM language reference manual," 2015).

2.2 Languages for Functional Optimizations

Compilers for functional languages sometimes use or extend representations from the imperative world. However, especially for high-level optimization, it is more common to employ a simpler functional language, in much the same way that the typical three-address code follows the imperative model.

The λ-Calculus

Three-address codes represent imperative programs by a minimum of constructs. Similarly, Church's λ-*calculus* (Barendregt, 1984) boils functional programs down to their essentials: functions, variables, and applications. A function is represented as $\lambda x.\, M$, where x is a variable and M is the function

$$\begin{array}{rl} \text{Variable:} & \qquad\qquad x, y, z, \ldots \\ \text{Term:} & M, N ::= x \mid M\,N \mid \lambda x.\, M \end{array}$$

FIGURE 4. The syntax of the untyped λ-calculus.

body in which x is bound; application of M to N is written simply as the juxtaposition $M\,N$. See Fig. 4.

An advantage of the λ-calculus is that its semantics can be given purely as a system of simplification rules, or *rewrite rules*, on the terms themselves. The crucial one is called the *β-rule*, which in its most general conventional form is

$$(\lambda x.\, M)\, N \Rightarrow M\{N/x\}$$

This says that to apply a known function $\lambda x.\, M$ to an argument N, you take the body M and *substitute* N for the occurrences of x. (We haven't yet said what terms are allowed as N or how to apply the rule on a subterm of a larger term; these are specified by the evaluation order.) Applying the β-rule is called *β-reduction.*

The other rule is the *α-rule*, which simply says that we can rename a variable bound by a λ without changing the term's meaning, so long as we do so consistently. Applying the α-rule is called *α-conversion.* Because they are subject to α-conversion, variables have local scope, in much the way they do in most programming languages. For instance, since $\lambda x.\, x$ and $\lambda y.\, y$ are equivalent by the α-rule, no program's behavior can depend on the choice of x or y.

This is the whole of the syntax of the plain untyped λ-calculus, but the language is already rich enough to have spawned a whole field of research. Indeed, the untyped λ-calculus is universal, that is, Turing-complete—it is expressive enough to encode any program we could want to. For use in a compiler, however, it would be impractical; besides the sheer awkwardness of, say, representing the number five as $\lambda s.\,\lambda z.\,s(s(s(s(s\,z))))$, as is conventional (Hinze, 2005), the encoding would lose the program's structure, offering little help to an optimizer that wishes to perform code motion safely and effectively.

Nonetheless, there is little we need to add. First, we need literals and primitives to represent arithmetic sensibly; these are no problem. We'll deal with literals and other constants shortly, and primitives can simply be preallocated variable names.

Second, it helps to have a way to declare local variables and functions, so we want a **let/in** form, including a recursive version **let rec**. Again, these could be encoded easily in terms of application, but at little gain at the price of lost information.[1] The precise semantics of **let** differs more widely than that of function application; the simplest form is a variant of the β-rule,

$$\mathbf{let}\, x = N \,\mathbf{in}\, M \Rightarrow M\{N/x\},$$

but again, the form of N may be restricted. Also, rather than substitute for all instances of x at once, one can wait for the value of x to be needed (Ariola, Felleisen, Maraist, Odersky, & Wadler, 1995). For **let rec**, in particular, one has to be careful; see, for instance, (Pierce, 2002, chapters 20–21) for a practical treatment.

More significantly, we want structured data and control, namely *data constructors* and *pattern matching*. A data constructor is a constant with a particular *arity*. We write C^n for an unknown data constructor with arity n,

[1]Arguably, we could go without **let rec** by using *fixpoint combinators* such as the Y-combinator, but explicit knowledge of which functions are recursive is often beneficial. For example, GHC is aggressive about inlining non-recursive functions because doing so cannot cause the optimizer to loop.

but we will often drop the superscript. Applying C^n to n arguments[2] packages the values as a tagged record. Pattern matching is then performed by a **case** statement such as this one:

$$\begin{array}{l}\textbf{case}\ M\ \textbf{of}\\ \quad C^n\ x_1\ \ldots\ x_n\ \ \to\ N\\ \quad D^m\ y_1\ \ldots\ y_m \to\ P\end{array}$$

This expression evaluates M to a constructor and its arguments, then checks whether the constructor is C or D, binds the corresponding variables to the arguments, and evaluates the chosen branch.

For example, the functional version of our array-sum example could be:

$$\begin{array}{lll} \mathit{sum} & = & \lambda xs.\ \textbf{case}\ xs\ \textbf{of}\\ & & \quad \mathit{Nil} \qquad\quad \to\ 0\\ & & \quad \mathit{Cons}\ x\ xs' \to\ x + \mathit{sum}\ xs' \end{array}$$

Here *Nil* is the empty list and *Cons x xs* prepends x onto the list xs. Note that *Nil* has arity zero, as do *True* and *False*. Zero-arity, or *nullary*, constructors thus act as constants. In fact, we can simply treat literals as special syntax for particular nullary constructors, so we don't need them as a separate construct.

The semantics of **case** is specified by a rule called the *case rule*:

$$\begin{array}{l}\textbf{case}\ C^n\ M_1\ \cdots\ M_n\ \textbf{of}\\ \quad \vdots \\ \quad C^n\ x_1\ \cdots\ x_n \to N \\ \quad \vdots\end{array} \quad \Rightarrow\ N\{M_1/x_1\}\cdots\{M_n/x_n\}$$

Hence, we reduce a **case** by finding the matching branch and substituting the arguments of the constructor.

[2] Like Haskell, we use *curried* constructors. For example, *Cons x xs* is *Cons* applied to the arguments x and xs. This avoids needing tuples as a separate construct.

We also allow a lone variable to act as a *wildcard pattern*:

$$\begin{array}{l} \textbf{case}\ C^n\ M_1\ \cdots\ M_n\ \textbf{of} \\ \quad \vdots \\ \quad x \to\ N \\ \quad \vdots \end{array} \quad \Rightarrow\ N\{C\ M_1\ \cdots\ M_n/x\} \quad \text{(if no other match)}$$

Languages such as ML and Haskell also allow for more complex patterns, but those can be re-expressed using these simple ones (Peyton Jones, 1987). Also, we don't need a separate **if** construct, since we can define:

$$\textbf{if}\ M\ \textbf{then}\ N\ \textbf{else}\ P\ \triangleq\ \begin{array}{l} \textbf{case}\ M\ \textbf{of} \\ \quad \mathit{True} \to N \\ \quad \mathit{False} \to P \end{array}$$

In all, we have the syntax in Fig. 5. Note that for compactness we may use braces and semicolons instead of separate lines in **case** and **let rec** expressions.

To make the language useful for writing or compiling programs, we need to say a bit more about the semantics than just the rewrite rules as we have seen them. The rules say what to do, but not in what order. Evaluation orders comprise a field of study all of their own (Plotkin, 1975; Ariola et al., 1995), but for our purposes, informal descriptions will suffice.

- In all practical languages, evaluation "stops at a lambda." Bodies of functions are not evaluated until they are called.

- In *call-by-value* languages, the arguments to a function are evaluated before the function is called. This amounts to restricting the β-rule:

$$(\lambda x.\,M)\,V \Rightarrow M\{V/x\}$$

Here V stands for a *value*, which must be a λ-abstraction, a constant, or a constructor applied to values. The restriction also affects **let** bindings—

the definition is evaluated before the body. Call-by-value **let rec** is usually allowed to define only λ-abstractions.[3]

– In *lazy* languages, function application occurs as soon as the function body is known, while the arguments are still unevaluated. Lazy **let** bindings similarly go unevaluated at first, and **let rec** needn't be restricted, allowing circular or (conceptually) infinite data structures. To make laziness practical, implementations use a *call-by-need* strategy (Ariola et al., 1995), which ensures that each value is still only evaluated once.

A related concept is *purity.* A language is called *pure* if evaluating an expression never has any *side effects*, such as overwriting a variable or performing I/O. An *impure* language is thus very sensitive to evaluation order—move the wrong function call, and `Hello World!` could become `World! Hello`. If the language is call-by-value, impurity is manageable, as one can predict from any function body the order in which terms will be evaluated. However, in a call-by-need language, if one writes $f\,M\,N\,P$, the order in which M, N, and P are evaluated (if at all) depends entirely on the body of f, which may not even be in the same module. Thus laziness practically necessitates purity.[4]

In an IL for optimization, the major impact of evaluation order and purity is the freedom with which terms can be rearranged. Of course, the purpose of the optimizer is often to *change* the order in which things are evaluated, but doing so in an impure language is hazardous. By contrast, the order in which

[3]In a call-by-value language, a variable always stands for a value. So how do we bind the value of a **let rec** inside its own definition while it's being computed? There are workarounds, such as using a mutable storage cell, but the need is not great enough to justify additional complication.

[4]It gets worse: Besides call-by-need, another class of non-strict languages is *parallel* languages. These evaluate the arguments and the body simultaneously. Thus, in the parallel setting, there is *no* defined order in which side effects in M, N, and P will run.

$$\begin{array}{rrl}
\text{Variable:} & & x, \ldots \\
\text{Constructor:} & & C^n, \ldots \\
\text{Pattern:} & P ::= & \_ \mid C^n\, x_1 \cdots x_n \\
\text{Term:} & M, N ::= & x \mid C^n \mid M\, N \mid \lambda x.\, M \\
& & \mid \mathbf{case}\, M\, \mathbf{of}\, P_1 \to N_1; \ldots; P_n \to N_n \\
& & \mid \mathbf{let}\, x = M\, \mathbf{in}\, N \\
& & \mid \mathbf{let\,rec}\, x_1 = M_1; \ldots; x_n = M_n\, \mathbf{in}\, N
\end{array}$$

FIGURE 5. A λ-calculus with data constructors, recursion, and **case**.

expressions appear in a lazy language often does not matter, so the compiler has a great deal of freedom.

Since it is so rigid, an impure call-by-value λ-calculus is cumbersome as an optimizing IL, even for call-by-value source languages (though it can serve as an “abstract source” language (Appel, 1992, chapter 4)). However, if the source language is lazy, it may happily be optimized using plain λ-terms—in fact, our λ-calculus is essentially an untyped version of GHC's Core language (Peyton Jones & Launchbury, 1991; Santos, 1995; Peyton Jones & Santos, 1998).

Continuation-Passing Style

One advantage of a three-address code as an IL for imperative programs is that everything is spelled out: Intermediate results are given names. Every aspect of control flow, including order of operations, is explicit. Manipulating code is made easier by the regularity, and the similarity to assembly language makes code generation a simple syntactic translation.

Continuation-passing style (Sussman & Steele, Jr., 1975), or CPS, is a way for the λ-calculus to play much the same role for functional programs. As proved formally by Plotkin (Plotkin, 1975), a term written in CPS effectively specifies its own evaluation order, and any λ-term can be translated into CPS

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &= \lambda k.\, k\, x \\
\mathcal{C}[\![M\, N]\!] &= \lambda k.\, \mathcal{C}[\![M]\!](\lambda f.\, \mathcal{C}[\![N]\!](\lambda x.\, f\, x\, k)) \\
\mathcal{C}[\![\lambda x.\, M]\!] &= \lambda k.\, k\, (\lambda x.\, \lambda k_1.\, \mathcal{C}[\![M]\!]k_1) \\
\mathcal{C}[\![C^0]\!] &= \lambda k.\, k\, C^0 \\
\mathcal{C}[\![C^1]\!] &= \lambda k.\, k\, (\lambda x_1.\, \lambda k_1.\, k_1\, (C^1\, x_1)) \\
\mathcal{C}[\![C^n]\!] &= \lambda k.\, k\, (\lambda x_1.\, \lambda k_1.\, k_1\, (\cdots (\lambda x_n.\, \lambda k_n.\, k_n\, (C^n\, x_1\, \cdots\, x_n)))) \\
\mathcal{C}\left[\!\!\left[\begin{array}{l} \textbf{case}\ M\ \textbf{of} \\ \quad P_1 \to N_1 \\ \quad\vdots \\ \quad P_n \to N_n \end{array}\right]\!\!\right] &= \lambda k.\, \mathcal{C}[\![M]\!]\left(\begin{array}{l} \lambda x.\ \textbf{case}\ x\ \textbf{of} \\ \qquad P_1 \to \mathcal{C}[\![N_1]\!]k \\ \qquad\vdots \\ \qquad P_n \to \mathcal{C}[\![N_n]\!]k \end{array}\right) \\
\mathcal{C}[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!] &= \lambda k.\, \mathcal{C}[\![M]\!](\lambda x.\, \mathcal{C}[\![N]\!]k) \\
\mathcal{C}\left[\!\!\left[\begin{array}{l} \textbf{let rec} \\ \quad f_1 = \lambda x_1.\, M_1 \\ \quad\vdots \\ \quad f_n = \lambda x_n.\, M_n \\ \textbf{in} \\ N \end{array}\right]\!\!\right] &= \begin{array}{l} \lambda k.\ \textbf{let rec} \\ \qquad f_1 = \lambda x_1.\, \lambda k_1.\, \mathcal{C}[\![M_1]\!]k_1 \\ \qquad\vdots \\ \qquad f_n = \lambda x_n.\, \lambda k_n.\, \mathcal{C}[\![M_n]\!]k_n \\ \quad \textbf{in} \\ \quad \mathcal{C}[\![N]\!]k \end{array}
\end{aligned}
$$

FIGURE 6. A call-by-value CPS transform.

using a *CPS transform.* As it happens, CPS gives a name to each intermediate value, just as a three-address code does. The correspondence to assembly language is not as clear, often necessitating a lower-level IL acting as an *abstract machine* (Appel, 1992, chapter 13). Nonetheless, λ-terms in CPS have proven a useful IL for optimizations on functional programs.

The CPS method is simple enough that we can demonstrate it on a simple language by constructing a "compiler" on paper. We take the source language to be the one in Fig. 5, given call-by-value semantics. The transform is given in Fig. 6. Note that we assume throughout that k, k_1, etc., are *fresh* (they don't clash with existing names); one could always use names not allowed in the source language to avoid trouble.

The CPS transform makes each term into a function taking a *continuation*, which specifies what to do with the value of the term once it's computed. Since the continuation is itself a λ-abstraction, the result of each computation ends up bound to a variable. This variable then represents either a register or a location in memory where the value is stored.

We will consider the CPS rules in turn. Since the source language is call-by-value, any variable x will be bound to a value that has already been computed; hence, in the variable case, there is nothing more to do and we can pass x directly to the continuation k.

The key to the CPS transform is the rule for function calls:

$$\mathcal{C}[\![M\,N]\!] = \lambda k.\,\mathcal{C}[\![M]\!](\lambda f.\,\mathcal{C}[\![N]\!](\lambda x.\,f\,x\,k))$$

Once the CPS term is applied to a continuation k, the first thing to do is to evaluate the function M. The continuation for M binds its result as f, then goes on to evaluate the argument N. That result is bound as x. Finally, we perform the function call proper by invoking f with argument x and the original continuation k.

Translating a λ-abstraction is straightforward, though the translated version takes an extra argument for the continuation with which to evaluate the body.

Constructors appear somewhat more involved, but they are really merely special cases of functions. It is instructive to derive the CPS form for a constructor applied to arguments[5]:

$$
\begin{aligned}
\mathcal{C}[\![C^2\, M\, N]\!] &= \lambda k.\, \mathcal{C}[\![C^2 M]\!](\lambda f.\, \mathcal{C}[\![N]\!](\lambda y.\, f\, y\, k)) \\
&= \lambda k.\, (\lambda k.\, \mathcal{C}[\![C^2]\!](\lambda f.\, \mathcal{C}[\![M]\!](\lambda x.\, f\, x\, k)))(\lambda f.\, \mathcal{C}[\![N]\!](\lambda y.\, f\, y\, k)) \\
&\Rightarrow \lambda k.\, \mathcal{C}[\![C^2]\!](\lambda f.\, \mathcal{C}[\![M]\!](\lambda x.\, f\, x\, (\lambda f.\, \mathcal{C}[\![N]\!](\lambda y.\, f\, y\, k)))) \\
&= \lambda k.\, (\lambda k_0.\, k_0\, (\lambda x_1.\, \lambda k_1.\, k_1(\lambda x_2.\, \lambda k_2.\, k_2(C^2\, x_1\, x_2)))) \\
&\qquad\qquad (\lambda f.\, \mathcal{C}[\![M]\!](\lambda x.\, f\, x\, (\lambda f.\, \mathcal{C}[\![N]\!](\lambda y.\, f\, y\, k)))) \\
&\Rightarrow^* \lambda k.\, \mathcal{C}[\![M]\!](\lambda x.\, \mathcal{C}[\![N]\!](\lambda y.\, k(C^2\, x\, y)))
\end{aligned}
$$

Once the dust settles, the procedure is to evaluate M, bind it as x, evaluate N, bind it as y, then apply the data constructor and return.

The other rules are routine: A **case** evaluates the scrutinee first, then the continuation chooses the branch. A **let** evaluates the bound term first, then the body. A **let rec** simply binds the function literals and moves on (as suggested earlier, we assume that a call-by-value **let rec** only ever binds λ-abstractions).

The IL we get from the CPS transform is the *CPS language* given in Fig. 7. All function calls must involve both a function and an argument (or two) that are *values*—compile-time constants, variables standing for intermediate results, and applications of data constructors to values.

As an example, consider again the functional analog of our array-sum code:

$$
\begin{aligned}
\mathit{sum}\ \mathit{xs} \;=\; & \textbf{case}\ \mathit{xs}\ \textbf{of} \\
& \quad \mathit{Nil} \qquad\quad \to\ 0 \\
& \quad \mathit{Cons}\ x\ \mathit{xs}' \to\ x\ +\ \mathit{sum}\ \mathit{xs}'
\end{aligned}
$$

Its CPS form (after some simplification) is:

[5]Application associates to the left, so C^2MN parses as $(C^2M)N$.

$$
\begin{array}{rrl}
\text{Variable:} & & x, f, k, \ldots \\
\text{Constructor:} & & C^n, \ldots \\
\text{Pattern:} & P ::= & x \mid C^n\, x_1 \, \cdots \, x_n \\
\text{Value:} & V, W ::= & x \mid \lambda x.\, M \mid \lambda x.\, \lambda k.\, M \mid C^n\, V_1 \, \cdots \, V_n \\
\text{Term:} & M ::= & V\, W \mid V\, W_1\, W_2 \\
 & & \mid \mathbf{case}\, x\, \mathbf{of}\, P_1 \to M_1; \ldots; P_n \to M_n \\
 & & \mid \mathbf{let\,rec}\, f_1 = \lambda x_1.\, M_1; \ldots; f_n = \lambda x_n.\, M_n\, \mathbf{in}\, N
\end{array}
$$

FIGURE 7. A CPS language for the λ-calculus in Fig. 5, as produced by the transform in Fig. 6.

$$
\begin{array}{rcl}
\mathit{sum} & = & \lambda xs.\, \lambda k.\, \mathbf{case}\; xs\; \mathbf{of} \\
 & & \qquad \mathit{Nil} \qquad\quad \to \; k\; 0 \\
 & & \qquad \mathit{Cons}\; x\; xs' \to \; \mathit{sum}\; xs'\; (\lambda y.\, k\; (x \; + \; y))
\end{array}
$$

Now *sum* takes a continuation k as an extra parameter. In the *Nil* case, the answer is immediately passed to the continuation. In the *Cons* case, we perform a recursive call. Since, in the original expression $x \; + \; \mathit{sum}\; xs$, the first computation that would be made is the recursive call $\mathit{sum}\; xs$, the code for that call is now at the top of the CPS term. Its continuation will take the result y from the the recursion, add x, and return to the caller. (Here we suppose that $+$ is a primitive in the CPS language and isn't called using a continuation.)

Administrative Normal Form

CPS is expressive but heavyweight. A less radical alternative with many, though not all (Kennedy, 2007), of its virtues is the *administrative normal form*, better known as A-normal form or simply ANF. An ANF term has names for all arguments, and its evaluation order is spelled out, but function calls are not rewritten as tail calls.

$$
\begin{array}{rrl}
\text{Variable:} & & x, f, \ldots \\
\text{Constructor:} & & C^n, \ldots \\
\text{Pattern:} & P ::= & x \mid C^n\, x_1 \cdots x_n \\
\text{Value:} & V, W ::= & x \mid \lambda x.\, M \mid C^n\, V_1 \cdots V_n \\
\text{Term:} & M, N ::= & V \mid \mathbf{let}\, x = V W\, \mathbf{in}\, M \\
& & \mid \mathbf{case}\, x\, \mathbf{of}\, P_1 \to M_1; \ldots; \mid P_n \to M_n \\
& & \mid \mathbf{let\,rec}\, f_1 = \lambda x_1.\, M_1; \ldots; f_n = \lambda x_n.\, M_n\, \mathbf{in}\, N \\
& & \mid \mathbf{let}\, x = V\, \mathbf{in}\, M
\end{array}
$$

FIGURE 8. The ANF language produced by the CPS language in Fig. 35 by an inverse CPS transform.

ANF emerges naturally by considering the *inverse* of the CPS transform (Flanagan, Sabry, Duba, & Felleisen, 1993). In order to see what reductions of the CPS term do to the original term, we can consider transforming a λ-term to CPS, performing some reductions, then transforming it back. Not all reductions in CPS terms are interesting, however: the CPS transform introduces many λ-abstractions into the program, and β-reduction on these functions doesn't correspond to any behavior of the original term. These reductions are called *administrative reductions.* In order to see what the CPS transform does from the perspective of the source language, then, we can consider translating a term to CPS, performing any administrative reductions we can, then translating back.

The net result is a language (Fig. 8) in which all nontrivial values have names, but which is free of the "noise" of administrative reductions. For instance, the ANF for the *sum* function, taken using the inverse transform of the simplified CPS term above, is:

$$\begin{array}{lll} \mathit{sum} & = & \lambda \mathit{xs}.\ \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\ & & \quad \begin{array}{lll} \mathit{Nil} & \rightarrow & 0 \\ \mathit{Cons}\ x\ \mathit{xs}' & \rightarrow & \mathbf{let} \\ & & \quad y \ = \ \mathit{sum}\ \mathit{xs}' \\ & & \mathbf{in} \\ & & x + y \end{array} \end{array}$$

Compared to the original code, the only additional syntax is the binding of y for the partial result from the recursive call.

Many algorithms don't make direct use of the continuation terms in CPS, and thus they apply equally well in ANF. For example, conversion to SSA (to be seen shortly) works broadly the same way (Chakravarty, Keller, & Zadarnowski, 2003).

Losing explicit continuations is not without cost, however. One downside is that CPS terms are *closed* under the β-rule, that is, applying the β-rule to a CPS term gives another CPS term, so inlining and substitution can be applied freely in CPS. In contrast, ANF requires some extra renormalization. More seriously, though ANF enjoys a formal correspondence to CPS, this correspondence gives no guarantees about important properties such as code size. We will return to this point in Section 2.4 when we introduce the sequent calculus.

Comparing functional and imperative approaches

The functional and imperative worlds often express the same concepts in different terms, such as *loops* vs. *recursion.* The same is true of ILs—what appear to be radically different approaches are often accomplishing the same things.

In this section, we use ANF for comparison due to its lightweight syntax, but everything applies to CPS as well.

Hoisting vs. Floating

Despite coming from such different programming traditions, functional representations and CFGs share much in common, and many optimizations apply in either case under different guises. For example, consider a variation of our array-sum code (Fig. 9a), where we sum the first k entries of an array starting at index i_0. The loop thus exits when $i \geq i_0 + k$. This being a three-address code, $i_0 + k$ is broken out into its own computation, which we've named h. Notice, however, that i_0 and k never change, and hence neither does h. So it is wasteful to calculate it again each time through the loop, and *loop-invariant hoisting* moves the assignment to h before the loop (Fig. 9b).

Now consider an ANF version (Fig. 9c). It uses a tail-recursive function in place of the block, but it's otherwise similar. Precisely the same effect is achieved by *let floating*—since h's definition has no free variables defined inside *loop*, we can float it outside (Fig. 9d). Note that, in both cases, we need to be careful that there are no side effects interfering.

Converting between functional and SSA

In the case of SSA, one can do more than show that certain algorithms accomplish the same goal. SSA, CPS, and ANF are *equivalent* in the sense that one can translate faithfully between them (Kelsey, 1995; Appel, 1998b).

Conceptually, it is unsurprising that these forms should be interderivable. The single-assignment property is fundamental to functional programming, and the correspondence between `goto`s and tail calls is well known (Reynolds, 1998). There remain two apparent differences:

Nested structure As usually understood, the SSA namespace is "flat:" though we insist that each variable is defined once, all blocks can see each

```
    $i \leftarrow i_0$
    $s \leftarrow 0$
  $loop$:
    $h \leftarrow i_0 + k$
    $c \leftarrow i - h$
    **ifge** $c$ **then** $done$
    $t \leftarrow a @ i$
    $s \leftarrow s + t$
    $i \leftarrow i + 1$
    **jump** $loop$
  $done$:
    **return** $s$
```

(a) In three-address code.

```
    $i \leftarrow i_0$
    $s \leftarrow 0$
    $h \leftarrow i_0 + k$
  $loop$:
    $c \leftarrow i - h$
    **ifge** $c$ **then** $done$
    $t \leftarrow a @ i$
    $s \leftarrow s + t$
    $i \leftarrow i + 1$
    **jump** $loop$
  $done$:
    **return** $s$
```

(b) After hoisting.

```
let rec
  $loop\ i\ s$ =
    let
      $h = i_0 + k$
      $c = i - h$
    in
    if $c \geq 0$ then
      $done\ s$
    else
      let
        $t\ = a @ i$
        $s' = s + t$
        $i' = i + 1$
      in
      $loop\ i'\ s'$
  $done\ s\ = s$
in
$loop\ i_0\ 0$
```

(c) In ANF.

```
let
  $h = i_0 + k$
in
let rec
  $loop\ i\ s$ =
    let
      $c = i - h$
    in
    if $c \geq 0$ then
      $done\ s$
    else
      let
        $t\ = a @ i$
        $s' = s + t$
        $i' = i + 1$
      in
      $loop\ i'\ s'$
  $done\ s\ = s$
in
$loop\ i_0\ 0$
```

(d) After let floating.

FIGURE 9. Two representations of a function to compute a partial sum of the integers in an array, starting at index i_0 and including k elements.

definition. In contrast, CPS and ANF are defined in terms of nested functions in languages that employ lexical scope. Thus we must ensure that each variable's definition remains visible at each of its occurrences after translation.

ϕ-nodes A ϕ-node has no obvious meaning in a functional language; we must somehow encode it in terms of the available constructs: functions, variables, applications, and definitions.

We will consider each of these points in turn. First we turn to scope. The scoping rule for a functional program is simple: each occurrence of a variable must be inside the body of its binding. This means something slightly different for λ-bound and **let**-bound variables, but the effect is the same.

SSA programs obey a similar invariant: each variable's definition *dominates* each of its uses (Appel, 1998a). A block b_1 dominates a block b_2 if each path in the CFG from the start of the program to b_2 goes through b_1 (Prosser, 1959). If we make each block a function, then, we should be able to satisfy the scoping invariant, so long as we can nest each block's function inside the functions for the block's dominators. For CPS, blocks will naturally translate into continuations. In ANF, we can simply use regular functions; functions used for control flow in this way are often called *join points*, about which there will be much to say in Section 2.4.

Fortunately, dominance does form a tree structure, which we can use directly as the nesting structure for the translated program. As it happens, this *dominance tree* is something already calculated in the process of efficient translation to SSA form (Cytron et al., 1991).

Figure 10 demonstrates the dominance tree for a somewhat tangled section of code (Fig. 10a). The CFG is shown in Fig. 10b and the dominance

```
$b_0$:
  $i_0 \leftarrow 0$
  $s_0 \leftarrow 0$
$b_1$:
  $i \leftarrow \phi(i_0, i_1)$
  $s \leftarrow \phi(s_0, s_3)$
  $c_1 \leftarrow i - n$
  **ifge** $c_1$ **then** $b_2$ **else** $b_6$
$b_2$:
  $c_2 \leftarrow i\%2$
  **ifz** $c_2$ **then** $b_3$ **else** $b_4$
$b_3$:
  $t_1 \leftarrow a@i$
  $t_2 \leftarrow t_1 * 2$
  $s_1 \leftarrow s + t_2$
  **jump** $b_5$
$b_4$:
  $t_3 \leftarrow a@i$
  $s_2 \leftarrow s + t_3$
  **ifz** $t_3$ **then** $b_7$ **else** $b_5$
$b_5$:
  $s_3 \leftarrow \phi(s_1, s_2)$
  $i_1 \leftarrow i + 1$
  **jump** $b_1$
$b_6$:
  $c_3 \leftarrow s - 100$
  **ifgt** $c_3$ **then** $b_7$ **else** $b_8$
$b_7$:
  $s_4 \leftarrow -1$
$b_8$:
  $s_5 \leftarrow \phi(s, s_4)$
  **return** $s_5$
```

(a) A program in SSA form.

$$\begin{tikzcd}
 & b_0 \arrow[d] & & & \\
 & b_1 \arrow[d] \arrow[drr] & & & \\
 & b_2 \arrow[dl] \arrow[dr] & & b_6 \arrow[d] \arrow[dr] & \\
b_3 \arrow[dr] & & b_4 \arrow[dl] \arrow[r] & b_7 \arrow[r] & b_8 \\
 & b_5 \arrow[uuu, bend left] & & &
\end{tikzcd}$$

(b) The CFG for (a).

$$\begin{tikzcd}
 & & b_0 \arrow[d, no head] & & \\
 & & b_1 \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] \arrow[drr, no head] & & \\
 & b_2 \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & b_6 & b_7 & b_8 \\
b_3 & b_4 & b_5 & &
\end{tikzcd}$$

(c) The dominance tree for (a).

FIGURE 10. Translating from SSA to CPS or ANF using the dominance tree.

tree in Fig. 10c. Notice the impact of the edge from b_4 to b_7. Since a dominator must lie on *every* path from b_0, b_6 does not dominate b_7 or b_8. If not for that edge, it would dominate both.

Now we tackle ϕ-nodes. As a programming-language construct, a ϕ-node is an oddity. It says what a variable's value should be based on the previous state of control flow.[6] If we are to reinterpret blocks as functions, then, each call should cause the ϕ-node to get a different value. The solution is, of course, that the ϕ-node should become a *parameter* to the function, transferring responsibility for the value from the block (function) to its predecessor (invoker).

This can be formally justified by altering the way we draw the CFG. Since each entry in a ϕ-node relates the block to one of its predecessors, in a sense the value "belongs" on the *edge* of the CFG, not a vertex (see Fig. 11). SSA and CPS then represent the same information, but with the edge labels in different places.

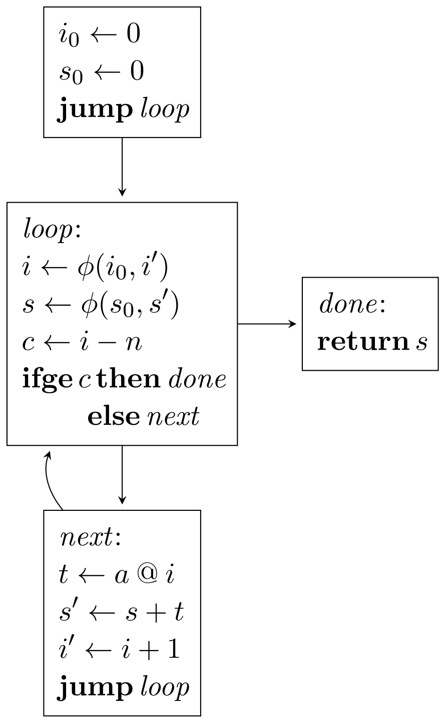
The result of translating Fig. 10a to ANF is shown in Fig. 12.

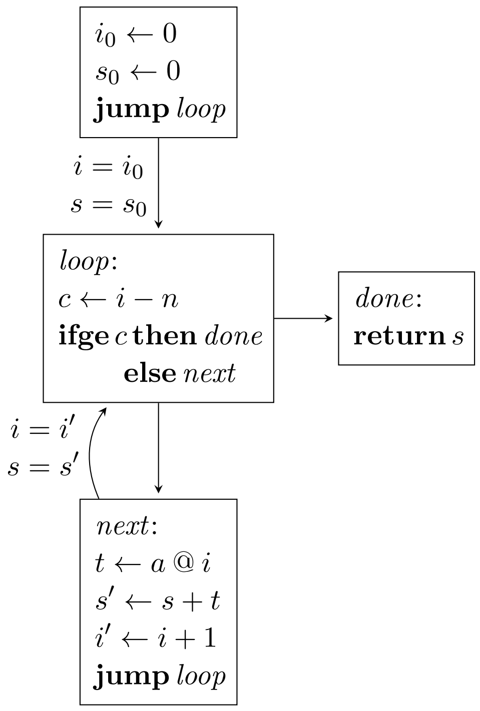
2.3 The Glasgow Haskell Compiler (GHC)

Implementing Lazy Evaluation

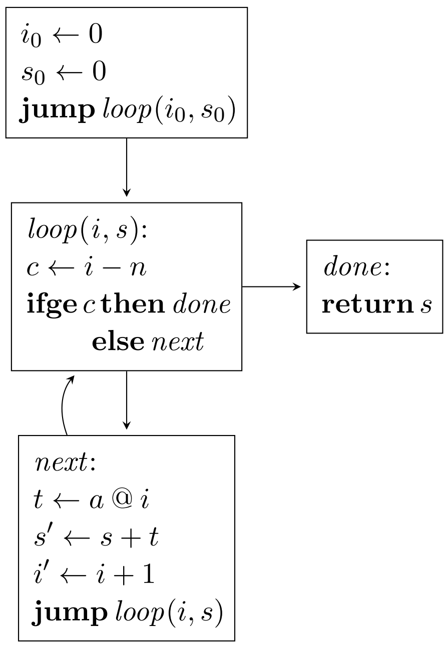
A compiler for any functional language has different concerns from those of an imperative language: higher-order functions and their closures are of paramount importance, interprocedural analysis is absolutely necessary, and alias analysis is an afterthought at most. But these are matters of emphasis rather than fundamental differences, as function application still works largely

[6]Oddly, this has been proposed before as a source language construct! A 1969 paper on optimization (Lowry & Medlock, 1969) suggested a novel form of declaration that would insinuate assignments to a variable at given line numbers in a program. Arguably, then, the ϕ-node was anticipated nearly twenty years beforehand, though of course its suitability in a source language is dubious. (To be sure, this was the era where `goto` was popular, so perhaps we should not be harsh.)





(b) The same, but moving ϕ-node values to the edges.



(c) A modified CFG form with parameterized labels, similar to continuations in CPS.

FIGURE 11. Modifying the CFG for Fig. 3 by moving the ϕ-node values into the predecessors.

```
λa. λn.
  let rec
    b₀ = let
              i₀ = 0
              s₀ = 0
         in
          let rec
            b₁ i s = let
                         c₁ = i − n
                     in
                      let rec
                        b₂    = let
                                    c₂ = i % 2
                                in
                                 let rec
                                   b₃    = let
                                               t₁ = a @ i
                                               t₂ = t₁ ∗ 2
                                               s₁ = s + t₂
                                           in
                                           b₅ s₁
                                   b₄    = let
                                               t₃ = a @ i
                                               s₂ = s + t₃
                                           in
                                            if t₃ = 0 then b₇ else b₅ s₂
                                   b₅ s₃ = let
                                               i₁ = i + 1
                                           in
                                           b₁ i₁ s₃
                                 in
                                  if c₂ = 0 then b₃ else b₄
                        b₆    = let
                                    c₃ = s − 100
                                in
                                 if c₃ > 0 then b₇ else b₈ s
                        b₇    = let
                                    s₄ = −1
                                in
                                b₈ s₄
                        b₈ s₅ = s₅
                      in
                       if c₁ ≥ 0 then b₂ else b₆
          in b₁ i₀ s₀
  in b₀
```

FIGURE 12. The program from Fig. 10a, in ANF.

the same way; it is merely that one expects a different sort of function to be common in an ML program from a C program.

Haskell is a more radical departure. Application is as important an operation as ever, but lazy evaluation turns it on its head. Variable lookup is put into particularly dramatic relief, as what was a single read could now be an arbitrary computation! Furthermore, this is not a benign change of execution model—a naïve implementation has a disastrous impact on performance.

The classical way to implement lazy evaluation is to use a *memo-thunk* (Hatcliff & Danvy, 1997; Okasaki, Lee, & Tarditi, 1994; Steele, Jr. & Sussman, 1976). This is a nullary function closure that will update itself when it finishes executing; on subsequent invocations, the new version will immediately return the cached answer. It is an effective strategy, and one might think that it merely shifts work from before a function is called to the first time its argument is needed. Unfortunately, this thought overlooks an important fact about modern hardware: an indirect jump, i.e., one to an address stored in memory rather than wired into the program, is *much* slower than a call to a known function, as it interferes with the pipelining and branch prediction that are crucial to performance. And in order to have a variable stand for a suspended computation, it must store the address of some code that will be executed, thus necessitating an indirect jump for each occurrence of each argument. Some improvements can be made—we can use tagged pointers to avoid an indirect call in the already-evaluated case, for instance (Marlow, Yakushev, & Peyton Jones, 2007)—but we cannot avoid at least one indirect function call per evaluated memo-thunk.

The GHC optimizer's fundamental task, then, is to *avoid lazy evaluation* as much as possible. Like polymorphism before it, laziness is a luxury for programming but a catastrophe for performance.

The Core Language

There were two important design principles (Peyton Jones & Santos, 1998) behind the design of Core:

1. *Provide an operational interpretation.* Given the severe penalties associated with lazy evaluation, GHC cannot afford to be blasé about fine details of evaluation order. Yet we would not want Core to be bogged down by operational details. Ideally, then, we want to choose constructs judiciously, so as to remain focused on the mission to eliminate laziness.

2. *Preserve type information.* Since type information is erased at run time, it is tempting to throw away types as soon as possible. But some passes, such as strictness analysis, can make good use of types if they are available. Perhaps more importantly, however, a typed IL can be of enormous use in developing the compiler itself by allowing an IL-level type checker to detect bugs early (Peyton Jones & Meijer, 1997): "It is quite difficult to write an incorrect transformation that is type correct."

   Though GHC does not, one can also use typed representations for formal verification. If the target language of a transform has a suitable type system, one can prove powerful faithfulness properties such as *full abstraction* (Plotkin, 1977), which means roughly that a translation does not expose implementation details that could not be observed in the source language. This is an important security property—the detail in question could be someone's password! For instance, a typed presentation of closure conversion (Minamide, Morrisett, & Harper, 1996) has been proved fully abstract (Ahmed & Blume, 2008), meaning not only that the conversion preserves meaning, but also that code written in the *target*

$$
\begin{array}{rrl}
\text{Variable:} & & x, y, z, \ldots \\
\text{Type Variable:} & & \alpha, \ldots \\
\text{Data Constructor:} & & C^n, \ldots \\
\text{Type:} & \tau ::= & \text{(see Fig. 14)} \\
\text{Pattern:} & P ::= & \_ \mid x \mid C^n\, x_1 \cdots x_n \\
\text{Term:} & M, N ::= & x \mid C^n \mid \lambda x : \tau.\, M \mid \Lambda \alpha.\, M \mid M\, N \mid M\, \tau \\
 & & \mid \mathbf{let}\, x : \tau = M\, \mathbf{in}\, N \\
 & & \mid \mathbf{let\,rec}\, x_1 : \tau_1 = M_1; \ldots ; x_n : \tau_n = M_n\, \mathbf{in}\, N \\
 & & \mid \mathbf{case}\, M\, \mathbf{of}\, P_1 \to M_1 \mid \cdots \mid P_n \to M_n
\end{array}
$$

FIGURE 13. The GHC Core language as of 2006 (GHC 6.6), before coercions were added.

language cannot "cheat" by inspecting a function's closure (perhaps to find a password). By employing a typed assembly language (Morrisett, Walker, Crary, & Glew, 1999), a whole compiler could, in principle, be proved fully abstract.

The first consideration led to refined semantics for **let** and **case** and to the use of *unboxed* types and values. The second consideration led to the use of the *polymorphic λ-calculus*, better known as System F.

Syntax

The syntax of Core is given in Fig. 13. Besides the types, there are few surprises at the syntactic level.

Semantics

The operational reading of a **let** is that it *allocates a thunk*. Thunks are also allocated for nontrivial function arguments.[7] A **case** of an expression M always forces the evaluation of M down to *weak head-normal form*, or WHNF, meaning that a pattern match is always done on a λ, a literal, or a constructor application.

The semantics of **let** and **case** allow GHC to reason about space usage and strictness. An evaluation that might otherwise remain suspended can be forced by putting it in a **case**, which doesn't necessarily actually perform any pattern matching—the pattern can simply be some x to bind the result of computation, just as in the CPS form for a call-by-value language.

For example, suppose we are compiling a function application $f(x + y)$, where it is known that f is a *strict function*—that is, one that is certain to force its argument to be evaluated. Since we know that $x + y$ will be forced, it would be wasteful to allocate a memo-thunk for it. We'd end up with a closure in the heap, only for it to overwrite itself with the result of the addition. Possibly worse, evaluating the thunk would entail an indirect call, when we know right now what the call will be! Much better, then, to force the evaluation early by writing $\mathbf{case}\, x + y\, \mathbf{of}\ z \to f\ z$.

Giving **let** and **case** these specific meanings relieves Core of needing any explicit constructs for dealing with memory or evaluation order, keeping the syntax very light.

Types

The Core type system is shown in Fig. 14. The basis of the language is System F, or more specifically System F_ω, otherwise known as the higher-order polymorphic λ-calculus. This system describes both *types*, such as *Int* and *Bool*, and type *constructors*, such as *Maybe* and *List*. *List* is not itself a datatype *per*

[7] Some older versions of GHC enforced an ANF-like restriction that arguments be atomic; in these versions of Core, **let**s were the *only* terms that allocated thunks. Even in current GHC, however, Core is translated into a lower IL called STG, which *does* have this restriction, so function arguments that aren't variables still wind up getting **let**-bound.

$$\begin{array}{rl}
\text{Type Variable:} & \quad\quad \alpha, \ldots \\
\text{Type Constructor:} & \quad\quad T, \ldots \\
\text{Type:} & \tau, \sigma ::= \alpha \mid T \mid \tau\,\sigma \mid \tau \to \sigma \mid \forall \alpha : \kappa.\, \tau \\
\text{Kind:} & \quad \kappa ::= \star \mid \# \mid \kappa_1 \to \kappa_2
\end{array}$$

FIGURE 14. Types in the GHC Core language.

se; it takes a parameter, so that *List Int* is a type. Thus types, themselves, have types, which are called *kinds*.

Kinds come in three varieties: The kind $\star$ is the kind of typical datatypes like *Int* and *Bool*. Arrow kinds $\kappa_1 \to \kappa_2$, like arrow types, describe type constructors: *List* and *Maybe* have kind $\star \to \star$. The kind # is particular to the Core type system; it is the kind of *unboxed* datatypes.

Unboxed Types

As mentioned above, a primary objective of Haskell optimization is to reduce laziness. To this end, GHC made an unusual choice for a lazy-language compiler by expressing true machine-level values in its high-level IL (Peyton Jones & Launchbury, 1991). This demonstrates that the difference between "high-level" and "low-level" is often a matter of design—if some aspect of execution on the target machine is absolutely crucial, it can be worth encoding that aspect at the high level.

Boxed types are those represented by a pointer to a heap object (often an unevaluated thunk) and include all types that appear in standard Haskell code. Unboxed types are represented by raw memory or registers. For instance, an $\mathit{Int}_{\#}$ is a machine-level integer, what C would call an `int`. This has two major ramifications:

1. A term of unboxed type must be computed *strictly*, rather than lazily. An $\mathit{Int}_{\#}$ is represented by an actual native integer, not a thunk, so there is no mechanism for lazily computing it.

   Therefore, an argument or **let**-binding *must* be a simple variable or literal (as in ANF), not an expression that performs work[8], since these constructs represent *suspended* (lazy) computations. Work returning an unboxed value must take place under a **case**, which as always serves to fix evaluation order.

2. Since types are (eventually) erased by the compiler, polymorphic functions rely on the uniform representation (pointer to heap object) of boxed types. Since unboxed types have different representations, they cannot be used as parameters to polymorphic types or functions.

   Core enforces this restriction by giving unboxed types the special kind $\#$. A type variable or type constructor must be of kind $\star$ or an arrow kind built from $\star$s (such as $(\star \to \star) \to \star$). Since *List* has kind $\star \to \star$ and $\mathit{Int}_{\#}$ has kind $\#$, then, $\mathit{List}\ \mathit{Int}_{\#}$ is a kind error.

The payoff is that, in many ways, *Int* is just like any other datatype, and the same optimizations that eliminate constructors for other datatypes work to keep arithmetic unboxed. For instance, this is how GHC defines the addition operator in Core:[9]

$$\mathbf{data}\ \mathit{Int} = I_{\#}\ \mathit{Int}_{\#}$$

[8]The actual invariant is a bit more subtle. *Some* expressions of unlifted type can be **let**-bound or passed as arguments, but only if they are known to evaluate quickly without side effects or possible errors. Such expressions are called *ok-for-speculation*, because there is no harm in executing them speculatively in the hopes of saving work later, say by moving them out of a loop.

[9]Actually, the operator belongs to a *type class* and is therefore defined for many types besides *Int*. This is, however, the implementation of $(+)$ for *Int*.

$$
\begin{array}{l}
(+) \quad :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
x + y = \ \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad I_{\#}\ x_{\#} \to\ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad I_{\#}\ y_{\#} \to\ \textbf{case}\ x_{\#} +_{\#} y_{\#}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad s_{\#} \to I_{\#}\ s_{\#}
\end{array}
$$

Here $+_{\#}$ is the primitive addition operator. The innermost **case** is necessary because $x_{\#} +_{\#} y_{\#}$ is a term of unboxed type and thus cannot be used directly as the argument to $I_{\#}$.

Now consider the optimization of the term $x + y + z$. A naïve interpretation would compute $x + y$, getting back a boxed integer s, then compute $s + z$. Thus the box created for s is thrown out immediately, performing unnecessary work and straining the garbage collector.

Let us see, then, what GHC does with it. Since (+) is so small, applications of it will always be inlined, so after inlining (that is, substituting and then β-reducing), we have:

$$
\begin{array}{l}
\quad (x + y) + z \\
\Rightarrow \left(\begin{array}{l}
\textbf{case}\ x\ \textbf{of} \\
\quad I_{\#}\ x_{\#} \to\ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad I_{\#}\ y_{\#} \to\ \textbf{case}\ x_{\#} +_{\#} y_{\#}\ \textbf{of} \\
\qquad\qquad\qquad\qquad s_{\#} \to I_{\#}\ s_{\#}
\end{array} \right) +\ z \\
\Rightarrow \textbf{case} \left(\begin{array}{l}
\textbf{case}\ x\ \textbf{of} \\
\quad I_{\#}\ x_{\#} \to\ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad I_{\#}\ y_{\#} \to\ \textbf{case}\ x_{\#} +_{\#} y_{\#}\ \textbf{of} \\
\qquad\qquad\qquad\qquad s_{\#} \to I_{\#}\ s_{\#}
\end{array} \right) \textbf{of} \\
\qquad I_{\#}\ a_{\#} \to\ \textbf{case}\ z\ \textbf{of} \\
\qquad\qquad I_{\#}\ z_{\#} \to\ \textbf{case}\ a_{\#} +_{\#} z_{\#}\ \textbf{of} \\
\qquad\qquad\qquad I_{\#}\ t_{\#} \to I_{\#}\ t_{\#}
\end{array}
$$

Next, we perform the *case-of-case transform*, to be explored in detail in Section 2.4. Briefly, consider what happens to evaluate this term: First we must evaluate the inner **case**, which will eventually yield the value $I_\# \; s_\#$. Thus it is $I_\# \; s_\#$ that will be scrutinized by the outer **case**, and we can rewrite the term to reflect this knowledge:

```
⇒ case $x$ of
      $I_\# \; x_\# \to$ case $y$ of
            $I_\# \; y_\# \to$ case $x_\# +_\# y_\#$ of
                  $s_\# \to$ case $I_\# \; s_\#$ of
                        $I_\# \; a_\# \to$ case $z$ of
                              $I_\# \; z_\# \to$ case $a_\# +_\# z_\#$ of
                                    $t_\# \to I_\# \; t_\#$
```

Note that the case-of-case transform has exposed the box-unbox sequence as a *redex*—**case** $I_\# \; s_\#$ **of** ... can be reduced at compile time. Thus we eliminate the **case** and substitute $s_\#$ for $a_\#$:

```
⇒ case $x$ of
      $I_\# \; x_\# \to$ case $y$ of
            $I_\# \; y_\# \to$ case $x_\# +_\# y_\#$ of
                  $s_\# \to$ case $z$ of
                        $I_\# \; z_\# \to$ case $s_\# +_\# z_\#$ of
                              $t_\# \to I_\# \; t_\#$
```

Now we have efficient three-way addition, the way one might write it by hand: unbox x and y; add them; unbox z; add to previous sum; box the result. Moreover, no special knowledge of $(+)$ was required; merely applying the same algorithms used with all Core code exposed and eliminated the gratuitous boxing.

Coercions and System F_C

A primary requirement of any typed IL is that it can embed the type system of the source language. As type systems become more sophisticated, including features such as dependent types (Xi & Pfenning, 1999; Bove, Dybjer, & Norell, 2009; Brady, 2013), one might see reason for concern: can we retain the benefits of typed ILs without making them unwieldy?

Fortunately, the answer appears to be "yes," at least so far. GHC's extensions to Haskell have begun to intermingle typing and computation with features such as generalized algebraic datatypes (Peyton Jones, Vytiniotis, Weirich, & Washburn, 2006; Xi, Chen, & Chen, 2003) and type families (Chakravarty, Keller, & Peyton Jones, 2005). These have been accomodated in Core by the addition of a much simpler extension, the *coercion* (Sulzmann, Chakravarty, Jones, & Donnelly, 2007). This extended λ-calculus is called System F_C.

The essential idea is that complex features of the type system can be handled entirely by the source-level typechecker, which annotates the generated Core with pre-calculated *evidence*, i.e., proofs showing that terms have the required types. These annotations take the form of type-safe *casts*

$$M \triangleright \gamma,$$

where M has some type τ and γ is a coercion of type[10] $\tau \sim \tau'$, proving that τ and τ' are equivalent—so far as Core is concerned, they are the same type.

[10]Somewhat confusingly, the literature sometimes refers to coercions as *types* whose *kinds* have the form $\tau \sim \tau'$. The distinction is not consequential.

Example: Well-Typed Expressions

For example, consider generalized algebric datatypes (GADTs), a popular extension to standard Haskell allowing datatypes to express constraints succinctly. The traditional example of a GADT is one for well-typed expressions:

```
data Expr a where
  Lit  :: a                                  → Expr a
  Plus :: Expr Int  → Expr Int               → Expr Int
  Eq   :: Expr Int  → Expr Int               → Expr Bool
  If   :: Expr Bool → Expr a    → Expr a     → Expr a
```

Because different constructors produce terms with different types, a nonsensical term like *Plus* (*Lit* 3) (*Lit True*) is a compile-time error. Even better, we can write a well-typed interpreter:

```
eval   :: Expr a → a
eval e = case e of
             Lit a      → a
             Plus x y   → eval x + eval y
             Eq x y     → eval x == eval y
             If b x y   → if eval b then eval x else eval y
```

Clearly the GADT representation is convenient, as even though expressions can denote different types, there is no need to tag the returned values or to check for error cases. However, the source-level typechecker's job is now trickier. Notice that this single **case** expression has *different types* in different branches: the *Plus* branch returns an *Int*, but the *Eq* branch returns a *Bool*. So the typechecker needs to keep track of types that change in different cases. Here it is only the return type that changes, but if *eval* took a second parameter of type a, that parameter's type would change as well.

We promised that the added complexity of GADTs could be isolated from the Core type system. GHC's typechecker keeps this promise by rewriting a GADT like *Expr* as a regular datatype:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr}\ a\ \textbf{where} \\
\quad \begin{array}{lllllll}
\mathit{Lit} & :: a & & & & & \to \mathit{Expr}\ a \\
\mathit{Plus} & :: \mathit{Expr}\ \mathit{Int} & \to \mathit{Expr}\ \mathit{Int} & \to a & \sim & \mathit{Int} & \to \mathit{Expr}\ a \\
\mathit{Eq} & :: \mathit{Expr}\ \mathit{Int} & \to \mathit{Expr}\ \mathit{Int} & \to a & \sim & \mathit{Bool} & \to \mathit{Expr}\ a \\
\mathit{If} & :: \mathit{Expr}\ \mathit{Bool} & \to \mathit{Expr}\ a & \to \mathit{Expr}\ a & & & \to \mathit{Expr}\ a
\end{array}
\end{array}
$$

We have kept the GADT syntax, but *Expr* is now a traditional datatype—its constructors all return *Expr* a, no matter what a is chosen to be. The caveat is that *Plus* requires a proof that a is *actually Int*, and similarly with *Eq*, so we still can't use *Plus* to create an *Expr Bool*.

Here is how *eval* is desugared into Core[11]:

$$
\begin{array}{lll}
\mathit{eval} & :: & \mathit{Expr}\ a\ \to\ a \\
\mathit{eval}\ e = & \multicolumn{2}{l}{\textbf{case}\ e\ \textbf{of}} \\
 & & \begin{array}{ll}
 \mathit{Lit}\ a & \to a \\
 \mathit{Plus}\ x\ y\ \gamma & \to (\mathit{eval}\ x + \mathit{eval}\ y) \triangleright \gamma^{-1} \\
 \mathit{Eq}\ x\ y\ \gamma & \to (\mathit{eval}\ x == \mathit{eval}\ y) \triangleright \gamma^{-1} \\
 \mathit{If}\ b\ x\ y & \to \textbf{if}\ \mathit{eval}\ b\ \textbf{then}\ \mathit{eval}\ x\ \textbf{else}\ \mathit{eval}\ y
 \end{array}
\end{array}
$$

Now the *Plus* and *Eq* branches can access the coercion γ stored in the *Expr*. In the *Plus* case, γ has type $a \sim \mathit{Int}$. Now, $\mathit{eval}\ x + \mathit{eval}\ y$ has type *Int*, and we must return an a, so γ's type is "backwards"—we need $\mathit{Int} \sim a$. But of course type equivalence is symmetric, so we can always take the inverse γ^{-1}. The *Eq* case is similar.

Typechecking is now straightforward. The only novelty is checking the coercions themselves, but this is not hard.

2.4 Sequent Calculus for GHC

As mentioned in Section 2.2, ANF is more concise than CPS, yet it is formally equivalent. Thus reasoning about the observable behavior of a term's CPS translation carries over to its ANF form (Flanagan et al., 1993). However, an optimizing compiler is concerned with much more than observable behavior—the equivalence can tell us only which transformations are *correct*, not which are *desirable*. Thus it is worth considering what we might be trading for the syntactic economy of ANF or plain λ-terms.

Case Floating and Join Points

When dealing with plain λ-terms, one important operation is called *case floating* (Santos, 1995; Peyton Jones & Santos, 1998). In general, if the first step of evaluating some term will be to evaluate a **case**, then the **case** can be brought to the top of the term. For instance, a **case** might return a function

[11] Technically, the **if** becomes a **case** in Core.

that will be applied immediately:

$$\left(\begin{array}{l}\textbf{case } b \textbf{ of} \\ \quad \mathit{True} \;\to\; \lambda y.\, x + y \\ \quad \mathit{False} \to\; \lambda y.\, x * y\end{array}\right) 5 \Rightarrow \begin{array}{l}\textbf{case } b \textbf{ of} \\ \quad \mathit{True} \;\to\; (\lambda y.\, x + y)\, 5 \\ \quad \mathit{False} \to\; (\lambda y.\, x * y)\, 5\end{array}$$

$$\Rightarrow \begin{array}{l}\textbf{case } b \textbf{ of} \\ \quad \mathit{True} \;\to\; x + 5 \\ \quad \mathit{False} \to\; x * 5\end{array}$$

As can be seen, the purpose of case floating is generally to bring terms together in the hope of finding a redex, i.e., an opportunity to perform a compile-time computation. Here, the **case** was always returning a λ, so moving the application inward lets the function call happen at compile time.

One possible concern is that the argument has been duplicated. What if this 5 were instead some large expression?

$$\left(\begin{array}{l}\textbf{case } b \textbf{ of} \\ \quad \mathit{True} \;\to\; \lambda y.\, x + y \\ \quad \mathit{False} \to\; \lambda y.\, x * y\end{array}\right) \langle \text{BIG} \rangle \Rightarrow\Rightarrow \begin{array}{l}\textbf{case } b \textbf{ of} \\ \quad \mathit{True} \;\to\; x + \langle \text{BIG} \rangle \\ \quad \mathit{False} \to\; x * \langle \text{BIG} \rangle \quad \text{-- Oops!}\end{array}$$

The β-reduction may be a Pyrrhic victory if it causes code size to explode—consider that the branches may, themselves, be **case** expressions, leading to exponential blow-up. The obvious solution, which GHC uses whenever a value is about to become shared, is to introduce a **let**-binding:

$$\begin{pmatrix}\textbf{case } b \textbf{ of} & & \\ \quad \mathit{True} & \rightarrow & \lambda y.\, x + y \\ \quad \mathit{False} & \rightarrow & \lambda y.\, x * y\end{pmatrix} \langle \text{BIG} \rangle \Rrightarrow \begin{array}{l}\textbf{let} \\ \quad \mathit{arg} \;=\; \langle \text{BIG} \rangle \\ \textbf{in} \\ \textbf{case } b \textbf{ of} \\ \quad \mathit{True} \;\rightarrow\; x + \mathit{arg} \\ \quad \mathit{False} \;\rightarrow\; x * \mathit{arg}\end{array}$$

The situation is more complex in a call-by-value language: if b were an expression with side effects, we would need to be careful that ⟨BIG⟩ is still not evaluated until afterward.

In contrast, CPS does not need **case** floating as a special case, and it will avoid the sharing issue even in a call-by-value language. Here is the call-by-value CPS form of our term:

$$\lambda k.\; \begin{pmatrix}\lambda k_1.\, \textbf{case } b \textbf{ of} & & \\ \qquad \mathit{True} & \rightarrow & k_1\,(\lambda y.\, \lambda k_2.\, k_2\,(x + y)) \\ \qquad \mathit{False} & \rightarrow & k_1\,(\lambda y.\, \lambda k_2.\, k_2\,(x * y))\end{pmatrix} (\lambda f.\, \mathcal{C}\,[\![\langle \text{BIG} \rangle]\!]\;(\lambda y.\, f\; y\; k))$$

The CPS form of the **case** expression is now applied to the continuation that serves to evaluate ⟨BIG⟩ and apply it as an argument to whichever function comes out of the **case**. A simple β-reduction moves that continuation's use into the branches, bringing the **case** to the top without any special rule:

$$
\begin{array}{l}
\lambda k.\,\textbf{let} \\
\qquad k_1 = \lambda f.\,\mathcal{C}\,[\![\langle\text{BIG}\rangle]\!]\ (\lambda y.\,f\ y\ k) \\
\quad \textbf{in} \\
\quad \textbf{case}\ b\ \textbf{of} \\
\qquad \mathit{True}\ \to\ k_1\,(\lambda y.\,\lambda k_2.\,k_2\,(x+y)) \\
\qquad \mathit{False} \to\ k_1\,(\lambda y.\,\lambda k_2.\,k_2\,(x*y))
\end{array}
$$

Here we have used a variation on β-reduction that **let**-binds the argument rather than substituting it.

Eliminating the redex, the way we did with the λ-term, is trickier but doable. As is, we would need to inline k_1 at both call sites, which would duplicate $\langle$BIG$\rangle$ all over again. But we can instead give $\langle$BIG$\rangle$ its own binding and (since f was chosen fresh) float it out:

$$
\begin{array}{ll}
 & \lambda k.\,\textbf{let} \\
 & \qquad \mathit{arg} = \lambda k.\,\mathcal{C}\,[\![\langle\text{BIG}\rangle]\!]\ k \\
 & \qquad k_1 \ \ = \lambda f.\,\mathit{arg}\,(\lambda y.\,f\ y\ k) \\
 & \quad \textbf{in} \\
 & \quad \textbf{case}\ b\ \textbf{of} \\
 & \qquad \mathit{True}\ \to\ k_1\,(\lambda y.\,\lambda k_2.\,k_2(x+y)) \\
 & \qquad \mathit{False} \to\ k_1\,(\lambda y.\,\lambda k_2.\,k_2(x*y)) \\
 & \lambda k.\,\textbf{let} \\
 & \qquad \mathit{arg} = \lambda k.\,\mathcal{C}\,[\![\langle\text{BIG}\rangle]\!]\ k \\
 & \quad \textbf{in} \\
\Rightarrow^* & \quad \textbf{case}\ b\ \textbf{of} \\
 & \qquad \mathit{True}\ \to\ \mathit{arg}\,(\lambda y.\,k\,(x+y)) \\
 & \qquad \mathit{False} \to\ \mathit{arg}\,(\lambda y.\,k\,(x*y))
\end{array}
$$

Since k_1 became small, we inlined it in the branches, and now those branches use x and arg directly, just as in the λ-term.

None of these procedures were specific to **case** statements; only floating and careful inlining were required. And floating can be performed aggressively on the CPS form, since there is no danger of changing evaluation order (Plotkin, 1975).

What about ANF? Remember that ANF is derived from CPS by performing *all* administrative reductions, then translating back to direct style. So the naïve ANF transform duplicates the context of any **case**. In practice, of course, implementations are smarter, avoiding administrative reductions that would duplicate too much code. The leftover continuations, which would have disappeared due to administrative reductions, are then called *join points*:

```
let
  $j \; = \; \lambda f.$ let
            $arg \; = \; \langle \mathrm{BIG} \rangle$
          in
            $f \; arg$
in
case $b$ of
  $True \to \; j \; (\lambda y.\, x + y)$
  $False \to \; j \; (\lambda y.\, x * y)$
```

Now, in CPS, every function call is a tail call and thus function application, including continuation invocation, is cheap. In ANF, however, function calls generally entail all the usual overhead[12], so if we don't treat j specially somehow, we will introduce that overhead in making j a function.

[12]It may seem from this discussion that ANF itself imposes function-call overhead compared to CPS. In fact, it merely makes implicit again what CPS expresses explicitly—a continuation closure represents the call stack, including the return pointer, as a λ-term. If code generation in turn uses the call stack to implement continuations, or if CPS code is translated back to direct style before code generation (Kennedy, 2007), then CPS vs. ANF has no lasting significance after the optimizer. Alternatively, the compiler can simply let the continuations be values like any other, using heap-allocated "stack frames" and forgoing the stack entirely.

Fortunately, j *is* special: like a continuation, it is only ever tail-called. Furthermore, it is only used in the context that defined it, and not under a λ-abstraction. Therefore *all calls to j will return to the same point.* Hence there is no need to push a stack frame with a return address in order to invoke it, and we can generate code for it as we would a continuation in a CPS IL.

So we can create join points when translating to ANF, and we can recognize join points during code generation so that calling them is efficient. Have we regained everything lost from CPS to ANF? Unfortunately, no. The problem is that *join points may not stay join points* during optimization.

Suppose our term, now in ANF with a join point, is part of a bigger expression:

```
let
  $g = \lambda x.$ let
          $j = \lambda f.$ let
                    $arg = \langle \text{BIG} \rangle$
                  in
                    $f\ arg$
        in
        case $b$ of
          $True \to j\ (\lambda y.\ x + y)$
          $False \to j\ (\lambda y.\ x * y)$
in
case $g\ 1$ of
  $\langle \text{HUGE} \rangle$
```

By doing the latter, SML/NJ easily implements the `call/cc` operator, which allows user code to access its continuation, with little overhead (Appel, 1998a, §5.9).

Now suppose that this is the only reference to g (it does not appear in $\langle\text{HUGE}\rangle$). Thus we want to inline the call $g\ 1$ so we can remove the binding for g altogether. We can start by performing the β-reduction:

$$
\begin{array}{l}
\textbf{case} \left(\begin{array}{l}
\textbf{let} \\
\quad j \ = \ \lambda f.\, \textbf{let} \\
\qquad\qquad\quad \mathit{arg} \ = \ \langle\text{BIG}\rangle \\
\qquad\qquad \textbf{in} \\
\qquad\qquad f\ \mathit{arg} \\
\textbf{in} \\
\textbf{case}\ b\ \textbf{of} \\
\quad \mathit{True} \ \rightarrow \ j\ (\lambda y.\, x + y) \\
\quad \mathit{False} \rightarrow \ j\ (\lambda y.\, x * y)
\end{array}\right) \textbf{of} \\
\quad \langle\text{HUGE}\rangle
\end{array}
$$

This term is no longer in ANF. To renormalize, first we float out the **let**:

$$
\begin{array}{l}
\textbf{let} \\
\quad j \ = \ \lambda f.\, \textbf{let} \\
\qquad\qquad\quad \mathit{arg} \ = \ \langle\text{BIG}\rangle \\
\qquad\qquad \textbf{in} \\
\qquad\qquad f\ \mathit{arg} \\
\textbf{in} \\
\textbf{case} \left(\begin{array}{l}
\textbf{case}\ b\ \textbf{of} \\
\quad \mathit{True} \ \rightarrow \ j\ (\lambda y.\, x + y) \\
\quad \mathit{False} \rightarrow \ j\ (\lambda y.\, x * y)
\end{array}\right) \textbf{of} \\
\quad \langle\text{HUGE}\rangle
\end{array}
$$

Then, as before, we need to perform case floating. This time we're doing the *case-of-case* transformation (Peyton Jones & Santos, 1998; Santos, 1995). It is similar to the previous "app-of-case" case, so we'll need to make another join point. (Incidentally, here GHC would need to make a join point as well; the trick it used earlier to float out $\langle\text{BIG}\rangle$ won't help.)

```
let
  j  = λf. let
               arg = ⟨BIG⟩
             in
             f arg
  j₂ = λz. case z of
               ⟨HUGE⟩
in
case b of
  True  → j₂ (j (λy. x + y))
  False → j₂ (j (λy. x ∗ y))
```

We're back in ANF, but notice that j is no longer a join point—it's now called in non-tail position. Thus the function-call overhead has crept back in.

Introducing Sequent Calculus

Clearly it is hazardous to represent join points as normal functions and expect to find them later still intact. Thus we would like a representation that treats them fundamentally differently. In particular, it would help to enforce syntactically the invariant that a join point must be tail-called. Needing to systematize the notion of "tail call" leads us to consider an encoding that makes control flow explicit, like CPS. CPS is syntactically heavy, however. More importantly, CPS makes *too much* control flow explicit. As was its initial purpose (Plotkin, 1975), CPS specifies syntactically the order in which expressions are evaluated, leaving nothing to the evaluation strategy. This would be cumbersome for the GHC optimization engine, which takes full advantage of Haskell's underspecified evaluation order. Also, since CPS encodes the evaluation strategy, there is a different transform for each evaluation strategy, whereas there is a *single* correspondence between λ-calculi and sequent

$$\begin{array}{rrl}
\text{Term Variable:} & & x, f, \ldots \\
\text{Cont. Variable:} & & k, j, \ldots \\
\text{Data Constructor:} & & C^n, \ldots \\
\text{Type:} & \tau ::= & \text{(see Fig. 14)} \\
\text{Coercion:} & \gamma ::= & \text{(omitted)} \\
\text{Pattern:} & P ::= & x \mid C^n\, x_1 \ldots x_n \\
\text{Term:} & M ::= & x \mid C^n \mid \lambda x : \tau.\, M \mid \Lambda\alpha : \kappa.\, M \mid \mu k : \tau.\, K \\
\text{Continuation:} & E ::= & k \mid M \cdot E \mid \tau \cdot E \mid \rhd\gamma \cdot E \\
 & & \mid \textbf{case of}\, P_1 \to K_1; \ldots ; P_n \to K_n \\
\text{Command:} & K ::= & \langle M \mid E \rangle \mid \textbf{let rec}\, B_1; \ldots ; B_n\, \textbf{in}\, K \\
 & & \mid \textbf{let}\, B\, \textbf{in}\, K \\
\text{Binding:} & B ::= & x = M \mid \textbf{cont}\, k = E
\end{array}$$

FIGURE 15. The syntax for Sequent Core.

calculi. In other words, the first step to using CPS in GHC would be to worry over *which CPS to use*, whereas sequent calculus lets us forget about evaluation order until later.

The *sequent calculus* was invented by Gerhard Gentzen in his study of logic, in order to prove properties of his other system, *natural deduction* (Gentzen, 1935). Decades later, it was realized that natural deduction is intimately related to the λ-calculus by what is now called the Curry–Howard isomorphism (Howard, 1980). More recently, there has been interest in the similar way that the sequent calculus can be seen as a programming language (Herbelin, 1995). We propose an IL called Sequent Core (Fig. 15), based on a lazy fragment of a sequent calculus, Dual System F_C, that incorporates the type system of System F_C.

The sequent calculus divides the expression syntax into three categories: *terms*, which produce values; *continuations*, which consume values; and *commands*, which apply values to continuations. Thus, computation is modeled as the *interaction* between a value and its context.

Most of the term forms are familiar. The novel one is the *μ-abstraction*, whose syntax is borrowed from Parigot's $\lambda\mu$-calculus (Parigot, 1992) describing control operators. It is written $\mu k : \tau. K$, meaning bind the continuation as k and then perform the command K. The μ-abstraction arises by analogy with CPS, which represents each term as a function of a continuation; hence, we distinguish continuation bindings. Keeping this distinction, as well as other syntactic restrictions compared to CPS, makes it simple to convert freely between Core and Sequent Core as needed.

The continuations comprise the observations that can be made of a term: we can apply an argument to it (either a value or a type argument); we can cast it using a coercion; we can perform case analysis on it; or we can return it to some context.

A command either applies a term to a continuation or allocates using a **let** binding. Either a term or a continuation may be **let**-bound, either recursively or non-recursively. Note that recursive continuations don't arise from translating Core, but just as join points could be recognized before, we can perform *contification* (Kennedy, 2007; Fluet & Weeks, 2001) to turn a consistently tail-called function (even a recursive one) into a continuation.

In Sequent Core, however, we do not risk accidentally "ruining" a continuation. Consider again our problematic term:

$$\left(\begin{array}{l}\textbf{case } b \textbf{ of} \\ \quad \mathit{True} \to \lambda y.\, x + y \\ \quad \mathit{False} \to \lambda y.\, x * y\end{array}\right) \langle \text{BIG} \rangle$$

Here it is as a term in Sequent Core:

$$\mu k.\ \left\langle \mu k_1.\ \left\langle b \,\middle|\, \begin{array}{l}\textbf{case of}\\ \quad \mathit{True} \to \langle \lambda y.\ \mu k_2.\ \langle (+) \mid x \cdot y \cdot k_2 \rangle \mid k_1 \rangle \\ \quad \mathit{False} \to \langle \lambda y.\ \mu k_2.\ \langle (*) \mid x \cdot y \cdot k_2 \rangle \mid k_1 \rangle \end{array} \right\rangle \,\middle|\, \langle \text{BIG} \rangle \cdot k \right\rangle$$

As before, we can share the application between the two branches as a join point. The lazy form of the μ_β rule for applying a continuation is:

$$\langle \mu k.\, K \mid E \rangle \Rightarrow \textbf{let}\,\textbf{cont}\ k = E\,\textbf{in}\,K$$

Thus we can perform a *μ_β-reduction* (renaming k_1 as j):

$$\begin{array}{l}\mu k.\,\textbf{let}\\ \qquad \textbf{cont}\,j \;=\; \langle \text{BIG} \rangle \cdot k\\ \quad \textbf{in}\\ \quad \left\langle b \,\middle|\, \begin{array}{l}\textbf{case of}\\ \quad \mathit{True} \to \langle \lambda y.\ \mu k_2.\ \langle (+) \mid x \cdot y \cdot k_2 \rangle \mid j \rangle \\ \quad \mathit{False} \to \langle \lambda y.\ \mu k_2.\ \langle (*) \mid x \cdot y \cdot k_2 \rangle \mid j \rangle \end{array} \right\rangle\end{array}$$

Now, what happens in the troublesome case-of-case situation in Sequent Core? Suppose, again, there is a larger context:

$$\begin{array}{l}\textbf{case}\ \left(\begin{array}{l}\textbf{case}\ b\ \textbf{of}\\ \quad \mathit{True} \to \lambda y.\, x + y\\ \quad \mathit{False} \to \lambda y.\, x * y\end{array} \right) \langle \text{BIG} \rangle\ \textbf{of}\\ \quad \langle \text{HUGE} \rangle\end{array}$$

Our simplified Sequent Core term becomes:

$$\mu k_0.\ \left\langle \left(\begin{array}{l}\mu k.\,\textbf{let}\\ \qquad \textbf{cont}\,j \;=\; \langle \text{BIG} \rangle \cdot k\\ \quad \textbf{in}\\ \quad \left\langle b \,\middle|\, \begin{array}{l}\textbf{case of}\\ \quad \mathit{True} \to \langle \lambda y.\ \mu k_2.\ \langle (+) \mid x \cdot y \cdot k_2 \rangle \mid j \rangle \\ \quad \mathit{False} \to \langle \lambda y.\ \mu k_2.\ \langle (*) \mid x \cdot y \cdot k_2 \rangle \mid j \rangle \end{array} \right\rangle\end{array} \right) \,\middle|\, \begin{array}{l}\textbf{case of}\\ \quad \langle \text{HUGE} \rangle\end{array} \right\rangle$$

Recall that GHC's case-of-case transform would be pulling the outer **case** into both branches of the inner **case**. It would make a join point to avoid duplicating ⟨HUGE⟩, but then it would still "ruin" j. But here, a simple μ_β-reduction, subsituting for k afterward, gives:

$$\begin{array}{l}
\mu k_0.\ \mathbf{let} \\
\qquad \mathbf{cont}\, j \;=\; \langle \mathrm{BIG} \rangle \cdot \ \mathbf{case\ of} \\
\qquad\qquad\qquad\qquad\qquad \langle \mathrm{HUGE} \rangle \\
\quad \mathbf{in} \\
\quad \left\langle b \,\middle|\, \begin{array}{l} \mathbf{case\ of} \\ \quad \mathit{True} \to \langle \lambda y.\ \mu k_2.\ \langle (+) \,|\, x \cdot y \cdot k_2 \rangle \,|\, j \rangle \\ \quad \mathit{False} \to \langle \lambda y.\ \mu k_2.\ \langle (*) \,|\, x \cdot y \cdot k_2 \rangle \,|\, j \rangle \end{array} \right\rangle
\end{array}$$

We pull the **case** into j, where it has a chance to interact with ⟨BIG⟩, perhaps by matching a known constructor.

Observe that this would be a very unnatural code transformation on Core: normally, it would never make sense to move an outer context into some **let**-bound term. If nothing else, it would change the return type of j, from that of ⟨BIG⟩ to that of the branches ⟨HUGE⟩, which a correct transformation rarely does. But the invariants of Sequent Core make case-of-case a simple substitution like any other. In particular, since continuations are only typed according to their *argument* type, j *has no* "return type," so substituting the outer context for k preserves types. Also, invariants about how k must be used (see below) ensure that we haven't changed the outcome of any code path.

Type and Scope Invariants

A command is simply some code that runs; it has no type of its own. A well-typed command is simply one whose term and continuation have the same type. Similarly, a continuation takes its argument and runs, so it doesn't have an "outgoing type" any more than a term has an "incoming type."

This may be worrisome—have we allowed *control effects* into our language? Haskell (and hence Core) is supposed to be a lazy language whose evaluation order is loosely specified, yet so far, it seems we would allow terms that discriminate according to evaluation order. For example:

$$\begin{array}{l}\mu k.\,\mathbf{let}\\ \qquad \mathbf{cont}\,j_1 = \ \mathbf{case\,of}\ \{\ \_ \to \langle\text{“Left first”}\,|\,k\rangle\ \}\\ \qquad \mathbf{cont}\,j_2 = \ \mathbf{case\,of}\ \{\ \_ \to \langle\text{“Right first”}\,|\,k\rangle\ \}\\ \quad \mathbf{in}\\ \quad \langle (+)\,|\,(\mu k_1.\ \langle ()\,|\,j_1\rangle)\cdot(\mu k_2.\ \langle ()\,|\,j_2\rangle)\cdot k\rangle\end{array}$$

In this term, whichever operand to (+) is evaluated first will pass a string directly to the continuation k, interrupting the whole computation and revealing the evaluation order. We have already seen how freely GHC rearranges terms because such changes are not supposed to be observable to Haskell programs; thus allowing programs such as this would be disastrous. If nothing else, the above term has no counterpart in Core, suggesting that the difference between Core and Sequent Core is so profound as to require rethinking the entire optimization pipeline.

On the other hand, we do not want to compromise flexibility. A rule such as “k must occur free in each branch” would disallow having different branches return through different join points. Selective inlining and known-case optimizations can cause branches to diverge dramatically. Indeed, there is nothing objectionable about this term, which is superficially similar to the one above:

$$\begin{array}{l}\mu k.\,\mathbf{let}\\ \qquad \mathbf{cont}\,j_1 = \ \mathbf{case\,of}\ \{\ \_ \to \langle\text{“It was true”}\,|\,k\rangle\ \}\\ \qquad \mathbf{cont}\,j_2 = \ \mathbf{case\,of}\ \{\ \_ \to \langle\text{“It was false”}\,|\,k\rangle\ \}\\ \quad \mathbf{in}\\ \quad \left\langle b \,\middle|\, \begin{array}{l}\mathbf{case\,of}\\ \quad \mathit{True} \to \langle ()\,|\,j_1\rangle\\ \quad \mathit{False} \to \langle ()\,|\,j_2\rangle\end{array}\right\rangle\end{array}$$

The solution is simple, as suggested by (Kennedy, 2007). We impose the scoping rule that *continuation variables may not appear free in terms.* Thus, we say terms are *continuation-closed.* This forbids j_1 and j_2 from being invoked from argument terms, where they constitute an impure computation, but not from **case** branches, where they are properly used to describe control flow.

It should be clear, then, that if the evaluation of the term $\mu k.\, K$ completes normally, k *must* be invoked. An informal proof: If K has no **let cont** bindings, then invoking k is "the only way out."[13] If there *is* a local continuation (i.e., a join point) declared, then *it* can only recurse to itself or invoke k, and if it only recurses then computation does not complete normally. If there is a join point j_1 and then another join point j_2, then it may be that j_2 is invoked, but it must eventually defer to k or to j_1 (and thus eventually to k) or else loop forever; and so on. By induction, either execution fails, or it succeeds through k.

The "inevitability" of a μ-bound continuation makes translating from Sequent Core back into Core easy. If we see the command $\langle M \mid E \rangle$, we can say confidently that the normal flow of control passes to E, so we translate E as the *context* of M. That is, we translate it as a fragment of syntax to be wrapped around M. For instance, writing $\mathcal{D}$ (for "direct style") for the translation to Core, we have:

$$\mathcal{D}\left[\!\!\left[\left\langle M \;\middle|\; \begin{array}{l} \textbf{case of} \\ \quad \mathit{True} \to K_1 \\ \quad \mathit{False} \to K_2 \end{array} \right\rangle \right]\!\!\right] = \begin{array}{l} \textbf{case}\; \mathcal{D}[\![M]\!]\; \textbf{of} \\ \quad \mathit{True} \to \mathcal{D}[\![K_1]\!] \\ \quad \mathit{False} \to \mathcal{D}[\![K_2]\!] \end{array}$$

[13]Note that, in general, a command has the structure of a tree of **case** expressions with continuation variables at the leaves. Without **let cont**, there is no way to bring new continuation variables into scope, and no terms (including arguments in continuations) may have free continuation variables, so only k may occur free at all. There may be branches with *no* continuation variables, since an empty **case** statement is allowed (typically when it is known that a term crashes or loops, so its continuation is dead code). Hence the stipulation at the start that computation "computes normally."

$$
\begin{array}{ll}
\mathcal{D}\,[\![\langle M \mid E\rangle]\!]_k & = \mathcal{D}\,[\![E]\!]\;[\,\mathcal{D}\,[\![M]\!]_k]\\
\mathcal{D}\,[\![\mathbf{let}\ B\ \mathbf{in}\ K]\!]_k & = \mathbf{let}\,\mathcal{D}\,[\![B]\!]_k\ \mathbf{in}\,\mathcal{D}\,[\![K]\!]_k\\
\mathcal{D}\left[\!\!\left[\mathbf{let\,rec}\ \vec{B}\ \mathbf{in}\ K\right]\!\!\right]_k & = \mathbf{let\,rec}\ \overrightarrow{\mathcal{D}\,[\![B]\!]_k}\ \mathbf{in}\,\mathcal{D}\,[\![K]\!]_k\\
\\
\mathcal{D}\,[\![x\ =\ M]\!]_k & = x\ =\ \mathcal{D}\,[\![M]\!]\\
\mathcal{D}\,[\![\mathbf{cont}\ j\ =\ E]\!]_k & = j\ =\ \lambda x\ \to\ \mathcal{D}\,[\![E]\!]_k\ [x]\\
\\
\mathcal{D}\,[\![x]\!] & = x\\
\mathcal{D}\,[\![C]\!] & = C\\
\mathcal{D}\,[\![\lambda\,x.\,M]\!] & = \lambda\,x.\,\mathcal{D}\,[\![M]\!]\\
\mathcal{D}\,[\![\Lambda\,\alpha.\,M]\!] & = \Lambda\,\alpha.\,\mathcal{D}\,[\![M]\!]\\
\mathcal{D}\,[\![\mu\,k.\,K]\!] & = \mathcal{D}\,[\![K]\!]_k\\
\\
\mathcal{D}\,[\![k]\!]_k & = \square\\
\mathcal{D}\,[\![j]\!]_k & = j\,\square\ \ (j\ \neq\ k)\\
\mathcal{D}\,[\![M\ \cdot\ E]\!]_k & = \mathcal{D}\,[\![E]\!]_k\ [\,\mathcal{D}\,[\![M]\!]\ \square]\\
\mathcal{D}\,[\![\tau\ \cdot\ E]\!]_k & = \mathcal{D}\,[\![E]\!]_k\ [\tau\,\square]\\
\mathcal{D}\,[\![\triangleright\gamma\ \cdot\ E]\!]_k & = \mathcal{D}\,[\![E]\!]_k\ [\square\ \triangleright\ \gamma]\\
\mathcal{D}\left[\!\!\left[\mathbf{case\ of}\ \overrightarrow{P\ \to\ K}\right]\!\!\right]_k & = \mathbf{case}\,\square\,\mathbf{of}\ \{\ \overrightarrow{P\ \to\ \mathcal{D}\,[\![K]\!]_k}\ \}
\end{array}
$$

FIGURE 16. The readback translation $\mathcal{D}$ from Sequent Core back to Core, as defined on commands, bindings, terms, and continuations.

Thus, we can easily return to Core after working in Sequent Core. The full definition of $\mathcal{D}$ is given in Fig. 16. The extra argument k is carried so that, when translating $\mu k.\, K$, we can distinguish between the k that marks the current context from other continuation variables introduced as join points. (Since terms are continuation-closed, only one such k can be μ-bound at a time.)

There is some overhead in translating back and forth, but early experience suggests it is tolerable.

2.5 Join Points in Direct Style

While the prototype implementation demonstrated Sequent Core's feasibility, the sheer size and disruptiveness of the change posed major challenges in a production compiler. The central *Expr* datatype became three mutually-recursive datatypes for terms, continuations, and commands. Thus every module being adapted to the Sequent Core world had to have each pass rewritten, often with one loop turning into three. No mature project undertakes such disruption unless absolutely necessary—code review alone would have taken countless hours, to say nothing of the torrent of new bugs.

Thus it is worth seeking a more modest solution providing the same benefits. The fact that we can translate back and forth offers a clue: by seeing what is preserved in the round trip and what isn't, we may hope to find what we could add to Core so that we might perform the same optimizations on Core that we were performing on Sequent Core.

To illustrate the need for care, consider:

$$\langle\, \mu\, k.\, \mathbf{let\,cont}\, j \;=\; E_1\; \mathbf{in}\; K \;|\, E_2 \,\rangle \;\Rightarrow\; \mathbf{let\,cont}\, j \;=\; E_1\; \{E_2/k\}\; \mathbf{in}\; K\; \{E_2/k\}$$

This is a crucial fact, for it deals with the situation where we would be tempted to ruin a join point. If we translate both sides back to Core using $\mathcal{D}$ and apply a tempting "substitution lemma," however, we get something nonsensical (here E stands for an evaluation context):

$$E[\mathbf{let}\, j \;=\; M \;\mathbf{in}\; N] \;\Rightarrow\; \mathbf{let}\, j \;=\; E[M] \;\mathbf{in}\; E[N]$$

This would allow us to derive[14]:

$$\begin{aligned} &\mathit{even}\,(\,\mathbf{let}\; b \;=\; \mathit{True}\;\mathbf{in}\;\mathbf{if}\; b\;\mathbf{then}\;0\;\mathbf{else}\;1) \\ \Rightarrow\, &\mathbf{let}\; b \;=\; \mathit{even}\;\mathit{True}\;\mathbf{in}\;\mathit{even}\,(\,\mathbf{if}\; b\;\mathbf{then}\;0\;\mathbf{else}\;1) \end{aligned}$$

Not only is this wrong, but *even True* doesn't so much as typecheck.

What has gone wrong? Observe from Fig. 16 that $\mathcal{D}$ turns both normal **let** bindings and continuation bindings into Core **let** bindings. If we came from Sequent Core, our "rule" can't tell whether j came from a join point or not.

Worse, the putative substitution lemma fails, for reasons turning precisely on whether or not a given identifier is a join point. This is what we would like to write:

$$\mathcal{D}\,[\![K\{E/k\}]\!]_k \;\Leftrightarrow\; \mathcal{D}\,[\![E]\!]_k\,[\,\mathcal{D}\,[\![K]\!]_k]$$

(Here $\Leftrightarrow$ means equality up to reductions in either direction.) The intuition is that substituting a continuation for k corresponds to adding more context to the top level of a term. The "lemma" breaks down completely, however, when K is an invocation of a join point:

$$\begin{aligned} &\mathcal{D}\,[\![\langle M \,|\, j\rangle \;\{E/k\}]\!]_k \\ =\; &\mathcal{D}\,[\![\langle M \,|\, j\rangle]\!]_k \\ =\; &j\;\mathcal{D}\,[\![M]\!] \end{aligned}$$

[14]We can use $\mathit{even}\;\Box$ as an evaluation context because *even* is strict.

$$\not\Leftrightarrow \mathcal{D}\,[\![E]\!]_k\;[j\,\mathcal{D}\,[\![M]\!]]$$
$$= \mathcal{D}\,[\![E]\!]_k\;[\,\mathcal{D}\,[\![\langle M \mid j\rangle]\!]_k]$$

The problem is that the continuation k does not appear in the invocation of a join point in Sequent Core. This makes sense, since a join point is an alternate continuation that retains its own context. But it means that remembering what's a join point and what isn't is crucial for maintaining any connection between Core and Sequent Core strong enough to start deriving useful relations.

Hence, anticipating the syntax introduced in Chapter III, we define $\mathcal{D}'$ to be the same as $\mathcal{D}$ except:

$$\begin{aligned} \mathcal{D}'\,[\![\mathbf{cont}\,j\;=\;E]\!]_k &= \mathbf{join}\,j\;x\;=\;\mathcal{D}'\,[\![E]\!]_k\,[x] \\ \mathcal{D}'\,[\![j]\!]_k &= \mathbf{jump}\;j\;\square \end{aligned}$$

Practically, all we have done is added a Boolean flag to each binding. Yet this is very powerful information, as the rule

$$E[\mathbf{let\,join}\,j\;x\;=\;M\;\mathbf{in}\;N]\;\Rightarrow\;\mathbf{let\,join}\,j\;x\;=\;E[M]\;\mathbf{in}\;E[N]$$

is not only sound but the key to avoiding ruining join points. To deal with our trouble with invoking a join point, we will also have this rule:

$$E[\mathbf{jump}\;j\;M]\;\Rightarrow\;\mathbf{jump}\;j\;M$$

Together with the usual rules for case floating, we can now give a substitution lemma to make the desired rules derivable from $\mathcal{D}$:

Lemma 1. *For any Sequent Core command K and continuation E:*

$$\mathcal{D}'\,[\![K\{E/k\}]\!]_k\;\Leftrightarrow\;\mathcal{D}'\,[\![E]\!]_k\,[\,\mathcal{D}'\,[\![K]\!]_k]$$

Thus many of the rules in Chapter III were derived by translating from Sequent Core. Since we also translated the type system, which enforces the restrictions on join points, we know that the translated rules are sound. This is the key advantage of using translation as a strategy for language extension: delicate matters such as what exactly the constraints on join points are or how case floating can work with them can be handled simply by appealing to translation. Since the sequent calculus handles join points so naturally, the soundness of the rule can be proved easily, and then translation saves us the implementation trouble of using a brand-new intermediate language for optimization.

CHAPTER III

Improving Compilation of Haskell

Most of the text in this chapter comes from (Maurer, Downen, Ariola, & Peyton Jones, 2017), which was a collaboration with Paul Downen (UO), Zena M. Ariola (UO), and Simon Peyton Jones (MSR). I led the research and did most of the writing. I also performed the software development (namely the extension to GHC), under Simon's helpful guidance.

Consider this code, in a functional language:

```
if (if e1 then e2 else e3) then e4 else e5
```

Many compilers will perform a *commuting conversion* (Girard, Taylor, & Lafont, 1989), which naïvely would produce:

```
if e1 then (if e2 then e4 else e5)
      else (if e3 then e4 else e5)
```

Commuting conversions are tremendously important in practice (Sec. 3.1), but there is a problem: the conversion duplicates `e4` and `e5`. A natural countermeasure is to name the offending expressions and duplicate the names instead:

```
let { j4 () = e4; j5 () = e5 }
in if e1 then (if e2 then j4 () else j5 ())
         else (if e3 then j4 () else j5 ())
```

We describe `j4` and `j5` as *join points*, because they say where execution of the two branches of the outer `if` joins up again. The duplication is gone, but a new problem has surfaced: the compiler may allocate closures for locally-defined functions like `j4` and `j5`. That is bad because allocation is expensive. And it is tantalizing because all we are doing here is encoding control flow: it is plain as a pikestaff that the "call" to `j4` should be no more than a jump, with no allocation anywhere. That's what a C compiler would do! Some code generators can cleverly eliminate the closures, but perhaps not if further transformations intervene.

The reader of Appel's inspirational book (Appel, 1992) may be thinking *"Just use continuation-passing style (CPS)!"* When expressed over CPS terms, many classic optimizations boil down to β-reduction (i.e.function application), or arithmetic reductions, or variants thereof. And indeed it turns out that commuting conversions fall out rather naturally as well. But using CPS comes at a fairly heavy price: the intermediate language becomes more complicated, some transformations are harder or out of reach, and (unlike direct style) CPS commits to a particular evaluation order.

Inspired by Flanagan et al. (Flanagan et al., 1993), the reader may now be thinking *"OK, just use administrative normal form (ANF)!"* That paper shows that many transformations achievable in CPS are equally accessible in direct style. ANF allows an optimizer to *exploit* CPS technology without needing to *implement* it. The motto is: *Think in CPS; work in direct style.*

But alas, a subsequent paper by Kennedy shows that there remain transformations that are inaccessible in ANF but fall out naturally in CPS (Kennedy, 2007). So the obvious question is this: could we extend ANF in some way, to get all the goodness of direct style *and* the benefits of CPS? In this paper we say "yes!", making the following contributions:

– We describe a modest extension to a direct-style λ-calculus intermediate language, namely adding join points (Sec. 3.2). We give the syntax, type system, and operational semantics, together with optimising transformations.

- We describe how to infer which ordinary bindings are in fact join points (Sec. 3.3). In a CPS setting this analysis is called *contification* (Kennedy, 2007), but it looks rather different in our setting.
- We show that join points can be recursive, and that recursive join points open up a new and entirely unexpected (to us) optimization opportunity for fusion (Sec. 3.4). In particular, this insight fully resolves a long-standing tension between two competing approaches to fusion, namely stream fusion (Coutts, Leshchinskiy, & Stewart, 2007) and unfold/destroy fusion (Svenningsson, 2002).
- We give some metatheory in Sec. 3.5, including type soundness and correctness of the optimizing transformations. We show the safety of adding jumps as a control effect by establishing an equivalence with System F.
- We demonstrate that our approach works at scale, in a state-of-the-art optimizing compiler for Haskell, GHC (Sec. 3.6). As hoped, adding join points turned out to be a very modest change, despite GHC's scale and complexity. Like any optimization, it does not make every program go faster, but it has a dramatic effect on some.

Overall, adding join points to ANF has an extremely good power-to-weight ratio, and we strongly recommend it to any direct-style compiler. Our title is somewhat tongue-in-cheek, but we now know of no optimizing transformation that is accessible to a CPS compiler but not to a direct-style one.

3.1 Motivation and Key Ideas

We review compilation techniques for commuting conversions, to expose the challenge that we tackle in this paper. For the sake of concreteness we describe the way things work in GHC. However, we believe that the whole paper is equally applicable to a call-by-value language.

Case-of-Case Transformation

Consider these function definitions:

```
isNothing :: Maybe a -> Bool
isNothing x = case x of Nothing -> True
                        Just _  -> False


mHead :: [a] -> Maybe a
mHead ps = case ps of []    -> Nothing
                      (p:_) -> Just p


null :: [a] -> Bool
null as = isNothing (mHead as)
```

Here `null` is a simple composition of the library functions `isNothing` and `mHead`. When the optimizer works on `null`, it will inline both `isNothing` and `mHead` to yield:

```
null as = case (case as of []    -> Nothing
                           (p:_) -> Just p) of
            { Nothing -> True; Just _ -> False }
```

Executed directly, this would be terribly inefficient; if the argument list is non-empty we would allocate a result `Just p` only to immediately decompose it. We want to move the outer case into the branches of the inner one, like this:

```
null as = case as of
            []  -> case Nothing of Nothing -> True
```

```
                                         Just z  -> False
             p:_ -> case Just p  of Nothing -> True
                                         Just _  -> False
```

This is a commuting conversion, specifically the *case-of-case transformation.* In this example, it now happens that both inner `case` expressions scrutinize a data constructor, so they can be simplified, yielding

```
null as = case as of { [] -> True; _:_ -> False }
```

which is exactly the code we would have written for `null` from scratch.

GHC does a tremendous amount of inlining, including across modules or even packages, so commuting conversions like this are very important in practice: they are the key that unlocks a cascade of further optimizations.

Join Points

Commuting conversions have a problem, though: *they often duplicate the outer* `case`. In our example that was OK, but what about

```
 case (case v of { p1 -> e1; p2 -> e2 }) of
    { Nothing -> BIG1; Just x  -> BIG2 }
```

where `BIG1` and `BIG2` are big expressions? We do not want to duplicate these large expressions, or we would risk bloating the size of the compiled code, perhaps exponentially when `case` expressions are deeply nested (Lindley, 2005). It is easy to avoid this duplication by first introducing an auxiliary `let` binding:

```
 let { j1 () = BIG1; j2 x  = BIG2 } in
 case (case v of { p1 -> e1; p2 -> e2 }) of
   { Nothing -> j1 (); Just x  -> j2 x }
```

Now we can move the outer `case` expression into the arms of the inner `case`, without duplicating `BIG1` or `BIG2`, thus:

```
let { j1 () = BIG1; j2 x  = BIG2 } in
case v of
  p1 -> case e1 of Nothing -> j1 ()
                   Just x  -> j2 x
  p2 -> case e2 of Nothing -> j1 ()
                   Just x  -> j2 x
```

Notice that `j2` takes as its parameter the variable bound by the pattern `Just x`, whereas `j1` has no parameters[1].

Compiling Join Points Efficiently

We call `j1` and `j2` *join points* because you can think of them as places where control joins up again, but so far they are perfectly ordinary `let`-bound functions, and as such they will be allocated as closures in the heap. But that's ridiculous: all that is happening here is control flow splitting and joining up again. A C compiler would generate a jump to a label, not a call to a heap-allocated function closure!

So, right before code generation, GHC performs a simple analysis to identify bindings that can be compiled as join points. This identifies `let`-bound functions that will never be captured in a closure or thunk, and will only be tail-called with exactly the right number of arguments. (We leave the exact criteria for Sec. 3.3.) These join-point bindings do not allocate anything; instead a tail call to a join point simply adjusts the stack and jumps to the code for the join point.

[1]The dummy unit parameter is not necessary in a lazy language, but it is in a call-by-value language.

The case-of-case transformation, including the idea of using `let` bindings to avoid duplication, is very old; for example, both are features of Steele's Rabbit compiler for Scheme (Steele, 1978). In Rabbit the transformation is limited to booleans, but the discussion above shows that it generalizes very naturally to arbitrary data types. In this more general form, it has been part of GHC for decades (Peyton Jones & Santos, 1998). Likewise, the idea of generating different (and much more efficient) code for non-escaping `let` bindings is well established in many other compilers (Tolmach & Oliva, 1998; Reppy, 2002; Keep, Hearn, & Dybvig, 2021) as well as GHC.

Preserving and Exploiting Join Points

So far so good, but there is a serious problem with recognizing join points only in the back end of the compiler. Consider this expression:

```
case (let j x = BIG in
      case v of { A -> j 1; B -> j 2; C -> True } of
  { True  -> False; False -> True }
```

Here `j` is a join point. Now suppose we do case-of-case on this expression. Treating the binding for `j` as an ordinary `let` binding (as GHC does today), we move the outer `case` past the `let`, and duplicate it into the branches of the inner `case`, yielding

```
let j x = BIG in
case v of
  A -> case (j 1) of { True -> False; False -> True }
  B -> case (j 2) of { True -> False; False -> True }
  C -> case True  of { True -> False; False -> True }
```

The third branch simplifies nicely, but the first two do not. There are two distinct problems:

1. The binding for `j` is no longer a join point (it is not tail-called), so the super-efficient code generation strategy does not apply, and the compiler will allocate a closure for `j` at runtime. This happens in practice: we have cases in which GHC's optimizer actually *increases* allocation because it inadvertently destroys a join point.
2. Even worse, the two copies of the outer `case` now scrutinize an uninformative call like `(j 1)`. So the extra code bloat from duplicating the outer `case` is entirely wasted. And it's a huge lost opportunity, as we shall see.

So *it is not enough to generate efficient code for join points; we must identify, preserve, and exploit them.* In our example, if the optimizer *knew* that the binding for `j` is a join point, it could exploit that knowledge to transform our original expression like this:

```
let j x = case BIG of True  -> False
                      False -> True
in case v of
  A -> j 1
  B -> j 2
  C -> case True of { True -> False; False -> True }
```

This is much, much better than our previous attempt:

– The outer case has moved into the right-hand side of the join point, so it now scrutinizes `BIG`. That's good, because `BIG` might be a data constructor or a `case` expression (which would expose another case-of-case opportunity). So the outer `case` now scrutinizes the actual result of the expression, rather than an uninformative join-point call. That solves problem (2).

– The `A` and `B` branches do not mention the outer `case`, because it has moved into the join point itself. So `j` is still tail-called and remains an efficiently-compiled join point. That solves problem (1).

– The outer `case` still scrutinizes the branches that do not finish with a join point call, e.g. the `C` branch.

The Key Idea

Thus motivated, in the rest of this paper we explore the following very simple idea:

– Distinguish certain `let` bindings as *join-point bindings*, and their (tail-)call sites as *jumps*.

– Adjust the case-of-case transformation to take account of join-point bindings and jumps.

– In all the other transformations carried out by the compiler, ensure that join points remain join points.

Our key innovation is that, by recognising join points as a language construct, we both *preserve* join poins through subsequent transformations, and *exploit* join points to make other tansformations more effective. Next, we formalize this approach; subsequent sections develop the consequences.

3.2 System F_J: Join Points and Jumps

We now formalize the intuitions developed so far by describing System F_J, a small intermediate language with join points. F_J is an extension of GHC's Core intermediate language (Peyton Jones & Santos, 1998). We omit existentials, GADTs, and coercions (Sulzmann et al., 2007), since they are largely orthogonal to join points.

Terms

x	$\in$	Term variables	
j	$\in$	Label variables	
e, u, v	$::=$	$x \mid l \mid \lambda x{:}\sigma.\, e \mid e\, u$	
	$\mid$	$\Lambda a.\, e \mid e\, \varphi$	Type polymorphism
	$\mid$	$K\, \vec{\varphi}\, \vec{e}$	Data construction
	$\mid$	$\mathbf{case}\, e\, \mathbf{of}\, \overrightarrow{alt}$	Case analysis
	$\mid$	$\mathbf{let}\, vb\, \mathbf{in}\, v$	Let binding
	$\mid$	$\mathbf{join}\, jb\, \mathbf{in}\, u$	Join-point binding
	$\mid$	$\mathbf{jump}\, j\, \vec{\varphi}\, \vec{e}\, \tau$	Jump
alt	$::=$	$K\, \overrightarrow{x{:}\sigma} \to u$	Case alternative

Value bindings and join-point bindings

vb	$::=$	$x{:}\tau = e$	Non-recursive value
	$\mid$	$\mathbf{rec}\, \overrightarrow{x{:}\tau = e}$	Recursive values
jb	$::=$	$j\, \vec{a}\, \overrightarrow{x{:}\sigma} = e$	Non-recursive join point
	$\mid$	$\mathbf{rec}\, \overrightarrow{j\, \vec{a}\, \overrightarrow{x{:}\sigma} = e}$	Recursive join points

Answers

A	$::=$	$\lambda x{:}\sigma.\, e \mid \Lambda a.\, e \mid K\, \vec{\varphi}\, \vec{v}$

Types

a, b	$\in$	Type variables	
τ, σ, φ	$::=$	$a \mid \sigma \to \tau \mid \tau\, \varphi \mid \forall a.\, \tau$	
	$\mid$	T	Datatype

Frames, evaluation contexts, and stacks

F	$::=$	$\Box\, v \mid \Box\, \tau$	Application
	$\mid$	$\mathbf{case}\, \Box\, \mathbf{of}\, \overrightarrow{p \to u}$	Case scrutinee
	$\mid$	$\mathbf{join}\, jb\, \mathbf{in}\, \Box$	Join point
E	$::=$	$\Box \mid F[E]$	Evaluation contexts
s	$::=$	$\varepsilon \mid F : s$	Stacks

Tail contexts

L	$::=$	$\Box$	Empty unary context
	$\mid$	$\mathbf{case}\, e\, \mathbf{of}\, \overrightarrow{p \to L}$	Case branches
	$\mid$	$\mathbf{let}\, vb\, \mathbf{in}\, L$	Body of let
	$\mid$	$\mathbf{join}\, j\, \vec{a}\, \overrightarrow{x{:}\sigma} = L\, \mathbf{in}\, L'$	Join point, body
	$\mid$	$\mathbf{join\, rec}\, \overrightarrow{j\, \vec{a}\, \overrightarrow{x{:}\sigma} = L}\, \mathbf{in}\, L'$	Rec join points, body

Miscellaneous

C	$\in$	General single-hole term contexts	
Σ	$::=$	$\cdot \mid \Sigma, x{:}\sigma = v$	Heap
c	$::=$	$\langle e;\, s;\, \Sigma \rangle$	Configuration

FIGURE 17. Syntax of System F_J.

Syntax

System F_J is a simple λ-calculus language in the style of System F, with **let** expressions, data type constructors, and **case** expressions; its syntax is given in Fig. 17. System F_J is an explicitly-typed language, so all binders are typed, but in our presentation we will often drop the type annotations.

The join-point extension is highlighted in the figure and consists of two new syntactic constructs:

– A **join** binding that declares a join point. Each join point has a name, a list of type parameters, a list of value parameters, and a body.

– A **jump** expression that invokes a join point, passing all indicated arguments as well as an additional type argument (as discussed below).

Although we use curried syntax for **jump**s, join points are *polyadic*; partial application is not allowed.

Static semantics

The type system for System F_J is given in Fig. 18, where typeof gives the type of a constructor and ctors gives the set of constructors for a datatype.

The typing judgement carries two environments, Γ and Δ, with Δ binding join points. The environment Δ is extended by a **join** (rules JBind and RJBind) and consulted at a **jump**. Note that we rely on scoping conventions in some places: if $\Gamma; \Delta \vdash e : \tau$, then every variable (type or term) free in e or τ appears in Γ, and the symbols in Γ are unique. Similarly, every label free in e appears in Δ.

To enforce that jumps are not used as side effects, Δ is reset in every premise for a subterm *whose runtime context is not statically known.* For example, consider **join** $j\ x\ =$ RHS **in** f (**jump** j *True Int*). Here the context in which the **jump** is invoked is not statically known—in a lazy language it

$$\boxed{\Gamma; \Delta \vdash e : \tau}$$

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma; \Delta \vdash x : \tau}\ \textsc{Var} \qquad \frac{\mathsf{typeof}(K) = \forall \vec{a}.\, \vec{\sigma} \to T\ \vec{a} \qquad \overrightarrow{\Gamma; \varepsilon \vdash u : \sigma\{a/\varphi\}}}{\Gamma; \Delta \vdash K\ \vec{\varphi}\ \vec{u} : T\ \vec{\varphi}}\ \textsc{Con}$$

$$\frac{\Gamma, (x{:}\sigma); \varepsilon \vdash e : \tau}{\Gamma; \Delta \vdash \lambda(x{:}\sigma).\, e : \sigma \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma, a; \varepsilon \vdash e : \tau}{\Gamma; \Delta \vdash \Lambda a.\, e : \forall a.\, \tau}\ \textsc{TAbs}$$

$$\frac{\Gamma; \Delta \vdash e : \sigma \to \tau \quad \Gamma; \varepsilon \vdash u : \sigma}{\Gamma; \Delta \vdash e\ u : \tau}\ \textsc{App} \qquad \frac{\Gamma; \Delta \vdash e : \forall a.\, \tau}{\Gamma; \Delta \vdash e\ \varphi : \tau\{a/\varphi\}}\ \textsc{TApp}$$

$$\frac{(j{:}\forall \vec{a}.\, \vec{\sigma} \to \forall r.\, r) \in \Delta \quad \overrightarrow{\Gamma; \varepsilon \vdash u : \sigma\overrightarrow{\{a/\varphi\}}}}{\Gamma; \Delta \vdash \mathbf{jump}\ j\ \vec{\varphi}\ \vec{u}\ \tau : \tau}\ \textsc{Jump}$$

$$\frac{\Gamma; \varepsilon \vdash u : \sigma \quad \Gamma, x{:}\sigma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \mathbf{let}\, x{:}\sigma = u\, \mathbf{in}\, e : \tau}\ \textsc{VBind} \qquad \frac{\overrightarrow{\Gamma, \overrightarrow{x{:}\sigma}; \varepsilon \vdash u : \sigma} \quad \Gamma, \overrightarrow{x{:}\sigma}; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \mathbf{let\, rec}\, \overrightarrow{x{:}\sigma = u}\, \mathbf{in}\, e : \tau}\ \textsc{RVBind}$$

$$\frac{\Gamma, \vec{a}, \overrightarrow{x{:}\sigma}; \Delta \vdash u : \tau \quad \Gamma; \Delta, (j{:}\forall \vec{a}.\, \vec{\sigma} \to \forall r.\, r) \vdash e : \tau}{\Gamma; \Delta \vdash \mathbf{join}\, j\ \vec{a}\ \overrightarrow{x{:}\sigma} = u\, \mathbf{in}\, e : \tau}\ \textsc{JBind}$$

$$\frac{\overrightarrow{\Gamma, \vec{a}, \overrightarrow{x{:}\sigma}; \Delta, \overrightarrow{j{:}\forall \vec{a}.\, \vec{\sigma} \to \forall r.\, r} \vdash u : \tau} \quad \Gamma; \Delta, \overrightarrow{j{:}\forall \vec{a}.\, \vec{\sigma} \to \forall r.\, r} \vdash e : \tau}{\Gamma; \Delta \vdash \mathbf{join\, rec}\, \overrightarrow{j\ \vec{a}\ \overrightarrow{x{:}\sigma} = u}\, \mathbf{in}\, e : \tau}\ \textsc{RJBind}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash e : T\ \vec{\varphi} \quad \overrightarrow{\mathsf{typeof}(K) = \forall \vec{a}.\, \vec{\sigma} \to T\ \vec{a}} \quad \overrightarrow{\vec{\nu} = \vec{\sigma}\overrightarrow{\{a/\varphi\}}} \\ \overrightarrow{\Gamma, \overrightarrow{x{:}\nu}; \Delta \vdash u : \tau} \quad \mathsf{ctors}(T) = \{\vec{K}\}\end{array}}{\Gamma; \Delta \vdash \mathbf{case}\, e\, \mathbf{of}\, \overrightarrow{K\ \overrightarrow{x{:}\nu} \to u} : \tau}\ \textsc{Case}$$

FIGURE 18. Type system for System F_J.

depends on how f uses its argument—so it cannot be compiled to “adjust the stack and jump.” So j is not a valid join point. We exclude such terms by resetting Δ to ε when typechecking the argument in rule App.

Nevertheless, the typing of join points is a little bit more flexible than you might suspect. Consider this expression:

$$\left(\begin{array}{l} \mathbf{join}\, j\, x = \mathrm{RHS} \\ \mathbf{in\, case}\, v\, \mathbf{of}\ A \to \mathbf{jump}\ j\ \mathit{True}\ \mathit{C2C} \\ \qquad\qquad\quad B \to \mathbf{jump}\ j\ \mathit{False}\ \mathit{C2C} \\ \qquad\qquad\quad C \to \lambda c.\, c \end{array}\right) \texttt{'x'}$$

where $\mathit{C2C} = \mathit{Char} \to \mathit{Char}$. This is certainly well typed. A valid transformation is to move the application to `'x'` into *both* the body *and* the right hand side of the **join**, thus:

$$\begin{array}{l} \mathbf{join}\, j\, x = \mathrm{RHS}\ \texttt{'x'} \\ \mathbf{in} \left(\begin{array}{l} \mathbf{case}\, v\, \mathbf{of}\ A \to \mathbf{jump}\ j\ \mathit{True}\ \mathit{C2C} \\ \qquad\qquad B \to \mathbf{jump}\ j\ \mathit{False}\ \mathit{C2C} \\ \qquad\qquad C \to \lambda c.\, c \end{array}\right) \texttt{'x'} \end{array}$$

Now we can move the application into the branches:

$$\begin{array}{l} \mathbf{join}\, j\, x = \mathrm{RHS}\ \texttt{'x'} \\ \mathbf{in\, case}\, v\, \mathbf{of}\ A \to (\mathbf{jump}\ j\ \mathit{True}\ \mathit{C2C})\ \texttt{'x'} \\ \qquad\qquad\quad B \to (\mathbf{jump}\ j\ \mathit{False}\ \mathit{C2C})\ \texttt{'x'} \\ \qquad\qquad\quad C \to (\lambda c.\, c)\ \texttt{'x'} \end{array}$$

Should this be well typed? The jumps to j are not exactly tail calls, but they can (and indeed must) discard their context—here the application to `'x'`—and resume execution at j. We will see shortly how this program can be further transformed to remove the redundant applications to `'x'`, but the point here is that this intermediate program is still well typed, as reflected by the fact that Δ is not reset in the function part of an application (rule App).

The types given to join points themselves deserve some attention. A join point that binds type variables $\vec{a}$ and value arguments of types $\vec{\sigma}$ is given the type $\forall \vec{a}.\, \vec{\sigma} \to \forall r.\, r$ (rule JBind). The return type indicated, namely $\forall r.\, r$, is often written $\bot$, and it indicates a non-returning function: a function which does not actually return can be safely given any return value. This is similar to how Haskell's `error` function has type $\forall a.\, \mathit{String} \to a$. We have merely moved the universal quantification to the end for consistency with the **join** syntax, which does not (and must not[2]) bind this "return-type parameter."

So a join point's type does not reflect the value of its body, and a jump can have any type whatsoever. What then keeps a join point from returning arbitrary values? It is the JBind rule (or its recursive variant) that checks the right hand side of the join point, making sure it is the same as that of the entire **join** expression. Thus we cannot have

$$\mathbf{join}\, j = \texttt{"Gotcha!"}\, \mathbf{in}\, \mathbf{if}\, b\, \mathbf{then}\, \mathbf{jump}\; j\; \mathit{Int}\, \mathbf{else}\, 4$$

because j returns a *String* but the body of the **join** returns an *Int*. In short, the burden of typechecking has moved: whereas a function can be declared to return any type but can only be invoked in certain contexts, a join point can be invoked in any context but can only return a certain type.

[2]When we introduce the *abort* axiom (Sec.), it will need to change this type argument arbitrarily, which it can only safely do if the type is never actually used in the other parameters.

$$\boxed{\langle e;\, s;\, \Sigma\rangle \mapsto \langle e';\, s';\, \Sigma'\rangle}$$

$$\begin{array}{rcll}
\langle F[e];\, s;\, \Sigma\rangle & \mapsto & \langle e;\, F : s;\, \Sigma\rangle & (\mathit{push}) \\
\langle \lambda x.\, e;\, \square\, v : s;\, \Sigma\rangle & \mapsto & \langle e;\, s;\, \Sigma, x = v\rangle & (\beta) \\
\langle \Lambda a.\, e;\, \square\, \varphi : s;\, \Sigma\rangle & \mapsto & \langle e\{a/\varphi\};\, s;\, \Sigma\rangle & (\beta_\tau) \\
\langle \mathbf{let}\, \mathit{vb}\, \mathbf{in}\, e;\, s;\, \Sigma\rangle & \mapsto & \langle e;\, s;\, \Sigma, \mathit{vb}\rangle & (\mathit{bind}) \\
\langle x;\, s;\, \Sigma[x = v]\rangle & \mapsto & \langle v;\, s;\, \Sigma[x = v]\rangle & (\mathit{look}) \\
\left\langle \begin{array}{c} K\, \vec{\varphi}\, \vec{v}; \\ \mathbf{case}\, \square\, \mathbf{of}\, \overrightarrow{\mathit{alt}} : s; \\ \Sigma \end{array} \right\rangle & \mapsto & \begin{array}{l} \langle u;\, s;\, \Sigma, \overrightarrow{x = v}\rangle \\ \text{if } (K\, \vec{x} \to u) \in \overrightarrow{\mathit{alt}} \end{array} & (\mathit{case}) \\
\left\langle \begin{array}{c} \mathbf{jump}\; j\; \vec{\varphi}\; \vec{v}\; \tau; \\ s' \mathbin{+\!\!+} (\mathbf{join}\, \mathit{jb}\, \mathbf{in}\, \square : s); \\ \Sigma \end{array} \right\rangle & \mapsto & \begin{array}{l} \left\langle \begin{array}{c} \overrightarrow{u\{a/\varphi\}}; \\ \mathbf{join}\, \mathit{jb}\, \mathbf{in}\, \square : s; \\ \Sigma, \overrightarrow{x = v} \end{array} \right\rangle \\ \text{if } (j\; \vec{a}\; \vec{x} = u) \in \mathit{jb} \end{array} & (\mathit{jump}) \\
\left\langle \begin{array}{c} A; \\ \mathbf{join}\, \mathit{jb}\, \mathbf{in}\, \square : s; \\ \Sigma \end{array} \right\rangle & \mapsto & \langle A;\, s;\, \Sigma\rangle & (\mathit{ans})
\end{array}$$

FIGURE 19. Call-by-name operational semantics for System F_J.

Finally, the reader may wonder why join points are polymorphic (apart from the result type). In F_J as presented here, we could manage with monomorphic join points, but they become absolutely necessary when we add data constructors that bind existential type variables. We omitted existentials from this paper for simplicity, but they are very important in practice and GHC certainly supports them.

Operational Semantics

We give System F_J an operational semantics (Fig. 19) in the style of an abstract machine. A *configuration* of the machine is a triple $\langle e;\, s;\, \Sigma\rangle$ consisting of an expression e which is the current focus of execution; a *stack* s representing the current evaluation context (including join-point bindings); and a *heap* Σ of value bindings. The stack is a list of *frames*, each of which is an argument to apply, a case analysis to perform, or a bound join point (or recursive group).

Each frame is moved to the stack via the *push* rule. Most of the rules are quite conventional. We describe only call-by-name evaluation here, as rule *look* shows; switching to call-by-need by pushing an update frame is absolutely standard.

Note that only *value* bindings are put in the heap. Join points are stack-allocated in a frame: they represent mere code blocks, not first-class function closures. As expected, a jump throws away its context (the *jump* rule); it does so by popping all the frames from the stack to the binding (as usual, $+\!\!+$ stands for the concatenation of two stacks):

$$\begin{aligned}
&\left\langle \begin{array}{l}\mathbf{join}\, j\, x = x \\ \mathbf{in}\,\mathbf{case}\,(\mathbf{jump}\; j\; 2\; (\mathit{Int} \to \mathit{Bool}))\,3\,\mathbf{of}\ldots;\ \varepsilon;\ \varepsilon\end{array}\right\rangle \\
&\mapsto^{\star} \left\langle \begin{array}{c}\mathbf{jump}\; j\; 2\; (\mathit{Int} \to \mathit{Bool}); \\ \square\, 3 : \mathbf{case}\,\square\,\mathbf{of}\ldots : \mathbf{join}\, j\; x = x\;\mathbf{in}\,\square : \varepsilon; \\ \varepsilon\end{array}\right\rangle \\
&\mapsto \langle x;\ \mathbf{join}\, j\; x = x\;\mathbf{in}\,\square : \varepsilon;\ x = 2\rangle
\end{aligned}$$

Here three frames are pushed onto the stack: the join-point binding, the case analysis, and finally the application of 3 to the jump. Then the jump is evaluated, popping the latter two frames, replacing the term with the one from the join point, and binding the argument.

The *ans* rule removes a join-point binding from the context once an answer A (see Fig. 17) is computed; note that a well-typed answer cannot contain a jump, so at that point the binding must be dead code. Continuing our example:

$$\langle x;\ \mathbf{join}\, j\; x = x\;\mathbf{in}\,\square : \varepsilon;\ x = 2\rangle \mapsto^{\star} \langle 2;\ \varepsilon;\ x = 2\rangle$$

Optimizing Transformations

The operational semantics operates on closed configurations. An optimizing compiler, by contrast, must transform *open* terms. To describe

$$\boxed{e = e'}$$

$$\begin{array}{rcll}
(\lambda x{:}\sigma.\, e)\, v & = & \mathbf{let}\, x{:}\sigma = v\, \mathbf{in}\, e & (\beta) \\
(\Lambda a.\, e)\, \varphi & = & e\{a/\varphi\} & (\beta_\tau) \\
\mathbf{let}\, vb\, \mathbf{in}\, C[x] & = & \mathbf{let}\, vb\, \mathbf{in}\, C[v] & (\mathit{inline}) \\
 & & \quad \text{if } (x{:}\sigma = v) \in vb & \\
\mathbf{let}\, vb\, \mathbf{in}\, e & = & e & (\mathit{drop}) \\
\mathbf{join}\, jb\, \mathbf{in}\, L[\vec{e}, \mathbf{jump}\; j\; \vec{\varphi}\; \vec{v}\; \tau, \vec{e'}] & = & \mathbf{join}\, jb\, \mathbf{in}\, L[\vec{e}, \mathbf{let}\, \overrightarrow{x{:}\sigma = v}\, \mathbf{in}\, u\overrightarrow{\{a/\varphi\}}, \vec{e'}] & (\mathit{jinline}) \\
 & & \quad \text{if } (j\; \vec{a}\; \overrightarrow{x{:}\sigma} = u) \in jb & \\
\mathbf{join}\, jb\, \mathbf{in}\, e & = & e & (\mathit{jdrop}) \\
\mathbf{case}\, K\; \vec{\varphi}\; \vec{v}\; \mathbf{of}\; \overrightarrow{alt} & = & \mathbf{let}\, \overrightarrow{x{:}\sigma = v}\, \mathbf{in}\, e & (\mathit{case}) \\
 & & \quad \text{if } (K\; \overrightarrow{x{:}\sigma} \to e) \in \overrightarrow{alt} & \\
E[\mathbf{case}\, e\, \mathbf{of}\, \overrightarrow{K\; \vec{x} \to u}] & = & \mathbf{case}\, e\, \mathbf{of}\, \overrightarrow{K\; \vec{x} \to E[u]} & (\mathit{casefloat}) \\
E[\mathbf{let}\, vb\, \mathbf{in}\, e] & = & \mathbf{let}\, vb\, \mathbf{in}\, E[e] & (\mathit{float}) \\
E[\mathbf{join}\, jb\, \mathbf{in}\, e] & = & \mathbf{join}\, E[jb]\, \mathbf{in}\, E[e] & (\mathit{jfloat}) \\
E[\mathbf{jump}\; j\; \vec{\varphi}\; \vec{e}\; \tau] : \tau' & = & \mathbf{jump}\; j\; \vec{\varphi}\; \vec{e}\; \tau' & (\mathit{abort})
\end{array}$$

FIGURE 20. Common optimizations for System F_J.

possible optimizations, then, we separately develop a sound *equational theory* (Fig. 20), which lays down the "rules of the game" by which the optimizer is allowed to work. It is up to the optimizer to determine how to apply the rules to rewrite code. All the axioms carry implicit scoping restrictions to avoid free-variable capture. (For example, *drop* requires that nothing bound by vb occurs free in e.)

The β, β_τ, and *case* axioms are analogues of the similarly-named rules in the operational semantics. Since there is no heap, β and *case* create **let** expressions instead. Compile-time substitution, or *inlining*, is performed for values by *inline* and for join points by *jinline*. If a binding is inlined exhaustively, it becomes dead code and can be eliminated by the *drop* or *jdrop* axiom. Values may be substituted anywhere[3], which we indicate using a general single-hole context C in *inline*. Inlining of join points is a bit more delicate. A jump indicates *both* that we should execute the join point *and* that we should throw out the evaluation context up to the join point's declaration. Simply copying the body accomplishes the former but not the latter. For example:

$$\mathbf{join}\, j\ (x : \mathit{Int}) = x + 1\,\mathbf{in}\,(\mathbf{jump}\ j\ 2\ (\mathit{Int} \to \mathit{Int}))\,3$$

If we naïvely inline j here, we end up with the ill-typed term:

$$\mathbf{join}\, j\ (x : \mathit{Int}) = x + 1\,\mathbf{in}\,(2 + 1)\,3$$

Inlining is safe, however, if the jump is a tail call, since then there is no extra evaluation context to throw away. To specify the allowable places to inline a join point, then, we use a syntactic notion called a *tail context.* A tail context L (see Fig. 17) is a multi-hole context describing the places where a term may return to its evaluation context. Since $\square\,3$ is not a tail context, the *jinline* axiom fails for the above term.

The *casefloat*, *float*, and *jfloat* axioms perform commuting conversions. Of the three, *jfloat* is novel. It does the transformation we wanted to perform in Sec. to avoid destroying a join point. It relies on a simple meta-syntactic function $E[\cdot]$ to push E into a join-point binding:

$$E[j\ \vec{a}\ \vec{x} = u] \triangleq (j\ \vec{a}\ \vec{x} = E[u])$$
$$E[\mathbf{rec}\,\overrightarrow{j\ \vec{a}\ \vec{x} = u}] \triangleq (\mathbf{rec}\,\overrightarrow{j\ \vec{a}\ \vec{x} = E[u]})$$

[3]For brevity, we have omitted rules allowing inlining a recursive definition into the definition itself (or another definition in the same recursive group).

Consider again the example at the beginning of Sec. . With our new syntax, we can write it as:

$$\mathbf{case}\left(\begin{array}{l}\mathbf{join}\, j\ x = \mathrm{BIG} \\ \mathbf{in}\,\mathbf{case}\, v\,\mathbf{of}\ A \to \mathbf{jump}\ j\ 1\ \mathit{Bool} \\ \qquad\qquad\quad B \to \mathbf{jump}\ j\ 2\ \mathit{Bool} \\ \qquad\qquad\quad C \to \mathit{True}\end{array}\right)\mathbf{of}$$

$$\{\mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}\}$$

We can use *jfloat* to move the outer **case** into both the right hand side of the **join** binding and into its body; use *casefloat* to move the outer **case** into the branches of the inner **case**; use *abort* to discard the outer **case** where it scrutinizes a **jump**; and use *case* to simplify the C alternative. The result is just what we want:

$$\begin{array}{l}\mathbf{join}\, j\ x = \mathbf{case}\,\mathrm{BIG}\,\mathbf{of}\ \{\mathit{True} \to \mathit{False}; \mathit{False} \to \mathit{True}\} \\ \mathbf{in}\,\mathbf{case}\, v\,\mathbf{of}\ A \to \mathbf{jump}\ j\ 1\ \mathit{Bool} \\ \qquad\qquad\quad B \to \mathbf{jump}\ j\ 2\ \mathit{Bool} \\ \qquad\qquad\quad C \to \mathit{False}\end{array}$$

The commute Axiom

The left-hand sides of axioms *float*, *jfloat*, and *casefloat* enumerate the forms of a tail context. That suggests that the three axioms are all instances of a single more general (yet equivalent) form:

$$E[L[\vec{e}]] = L[\overrightarrow{E[e]}] \quad (\mathit{commute})$$

To apply *commute* (forward) is to move the evaluation context into *each* hole of the tail context. Since the tail context describes the places where something is

$$\boxed{e = e'}$$

$$\begin{array}{rcll}
\mathbf{let}\, f = \Lambda\vec{a}.\,\lambda\vec{x}.\,u \,\mathbf{in}\, L[\vec{e}\,] : \tau & = & \mathbf{join}\, j\ \vec{a}\ \vec{x} = u \,\mathbf{in}\, L[\overrightarrow{\mathsf{tail}_\rho(e)}] & (\mathit{contify}) \\
 & & \text{if } \rho(f\ \vec{a}\ \vec{x}) = \mathbf{jump}\ j\ \vec{a}\ \vec{x}\ \tau & \\
 & & \text{and } f \notin \mathrm{fv}(L), u : \tau & \\
\mathbf{let\,rec}\, \overrightarrow{f = \Lambda\vec{a}.\,\lambda\vec{x}.\,L[\vec{u}\,]} \,\mathbf{in}\, L'[\vec{e}\,] : \tau & = & \mathbf{join\,rec}\, \overrightarrow{j\ \vec{a}\ \vec{x} = L[\overrightarrow{\mathsf{tail}_\rho(u)}]} \,\mathbf{in}\, L'[\overrightarrow{\mathsf{tail}_\rho(e)}] & (\mathit{contify}_{\mathit{rec}}) \\
 & & \text{if } \overrightarrow{\rho(f\ \vec{a}\ \vec{x}) = \mathbf{jump}\ j\ \vec{a}\ \vec{x}\ \tau} & \\
 & & \text{and } \overrightarrow{f \notin \mathrm{fv}(\vec{L}), f \notin \mathrm{fv}(L')}, \overrightarrow{L[\vec{u}\,] : \tau} &
\end{array}$$

$$\begin{array}{rcll}
\mathsf{tail}_\rho(f\ \vec{\sigma}\ \vec{u}\,) & \triangleq & e\overrightarrow{\{a/\sigma\}}\overrightarrow{\{x/u\}} & \text{if } \rho(f\ \vec{a}\ \vec{x}) = e \text{ and } \mathrm{dom}(\rho) \cap \mathrm{fv}(\vec{u}\,) = \emptyset \\
\mathsf{tail}_\rho(e) & \triangleq & e & \text{if } \mathrm{dom}(\rho) \cap \mathrm{fv}(e) = \emptyset \\
\mathsf{tail}_\rho(e) & \triangleq & \text{undefined} & \text{otherwise}
\end{array}$$

FIGURE 21. Contification as a source-to-source transformation.

returned to the evaluation context, *commute* "substitutes" the context into the places where it is invoked.[4]

We can also derive new axioms succinctly using tail contexts. For example, our commuting conversions as written do quite a bit of code duplication by copying E arbitrarily many times (into each branch of a **case** and each join point). Of course, in a real implementation, we would prefer not to do this, so instead we might use a different axiom:

$$E[L[\vec{e}\,] : \tau] = \mathbf{join}\, j\ x = E[x] \,\mathbf{in}\, L[\overrightarrow{\mathbf{jump}\ j\ e\ \tau}]$$

This can be derived from *commute* by first applying *jdrop* and *jinline* backward.

3.3 Contification: Inferring Join Points

Not all join points originate from commuting conversions. Though the source language doesn't have join points or jumps, many **let**-bound functions can be converted to join points without changing the meaning of the program. In particular, if *every* call to a given function is a tail call, and we turn the calls

[4]In fact, from a CPS standpoint, *commute* is *precisely* a substitution operation.

into jumps, then whenever one of the jumps is executed, there will be nothing to drop from the evaluation context (the s' in the *jump* rule will be empty).

The process is a form of *contification* (Kennedy, 2007) (or *continuation demotion*), which we describe in Fig. 21, where $\mathrm{fv}(e)$ means the set of free variables of e (and similarly $\mathrm{fv}(L)$ for tail contexts), and $\mathrm{dom}(\rho)$ means the domain of the environment ρ (to be described shortly).

The non-recursive version, *contify*, attempts to decompose the body of the **let** (i.e.the scope of f) into a tail context L and its arguments, where the arguments contain all the occurrences of f, then attempts to run the special partial function tail on each argument to the tail context. This function will only succeed if there are no non-tail calls to f.

The tail function takes an environment ρ mapping applications of contifiable variables f to jumps to corresponding join points j. For each expression that matches the form of a saturated call to such an f, then, tail turns the call into a jump to its j, *provided* that none of the arguments to the function contains a free occurrence of a variable being contified—an occurrence in argument position is disallowed by the typing rules. For any other expression, tail changes nothing but does check that no variable being contified appears; otherwise, tail fails, causing the *contify* axiom not to match.

There is one last proviso in the *contify* and *contify*$_{rec}$ axioms, which is that the body of each function to be contified must have the same type as the body of the **let**. This can fail to occur if some function f is polymorphic in its return type (Downen, Maurer, Ariola, & Peyton Jones, 2016).

Finding bindings to which *contify* or *contify*$_{rec}$ will apply is not difficult. Our implementation is essentially a free-variable analysis that also tracks whether each free variable has appeared *only* in the holes of tail contexts. This is much simpler than previous contification algorithms because we *only look* *for tail calls.* We invite the reader to compare to (Fluet & Weeks, 2001) or to Sec. 5 of (Kennedy, 2007), which both allow for more general calls to be dealt with. Yet we claim that, in concert with the simplifier and the Float In pass, our algorithm covers most of the same ground. To demonstrate, a convenient point of comparison is the *local CPS transformation* in Moby (Reppy, 2002), which produces mutually tail-recursive functions to improve code generation in much the same way GHC does. Note that Moby uses a direct-style intermediate representation, though its contification pass is expressed in terms of a CPS transform.

In essence, the final effect of Moby's local CPS transform is to turn

```
let f x = ...
in E[... f y ... f z ...]
```

(where the calls to `f` are tail calls within `E`) into

```
let { j x = E[x]; f x = j <rhs> }
in ...f y...f z...
```

where the tail calls to `f` are now compiled as efficient jumps. Note that `f` now matches the *contify* axiom, but it did not before because of the E in the way. Nonetheless, our extended GHC achieves the same effect as Moby, only in stages. Starting with:

$$\textbf{let}\, f\, x = \text{rhs}\, \textbf{in}\, E[\dots f\, y \dots f\, z \dots]$$

First, applying *float* from right to left floats f inward:

$$E[\textbf{let}\, f\, x = \text{rhs}\, \textbf{in}\, \dots f\, y \dots f\, z \dots]$$

Next, *contify* applies, since the calls to f are now tail calls:

$$E[\textbf{join}\, f\ x = \text{rhs}\,\textbf{in}\ \ldots \textbf{jump}\ f\ y\ \tau \ldots \textbf{jump}\ f\ z\ \tau \ldots]$$

And now *jfloat* pushes E into the join point f and the body:

$$\textbf{join}\, f\ x = E[\text{rhs}]\,\textbf{in}\ \ldots E[\textbf{jump}\ f\ y\ \tau] \ldots E[\textbf{jump}\ f\ z\ \tau] \ldots]$$

From here, *abort* removes E from the jumps, and we can abstract E by running *jdrop* and *jinline* backward:

$$\textbf{join}\,\{j\,x = E[x]; f\,x = \textbf{jump}\ j\ \text{rhs}\ \tau\}\,\textbf{in} \ldots f\,y \ldots f\,z \ldots$$

Thus we achieve the same result without any extra effort[5].

Naturally, contification is more routine and convenient in CPS-based compilers (Fluet & Weeks, 2001; Kennedy, 2007). The ability to handle an intervening context comes nearly "for free" since contexts already have names. Notably, it is still possible to name contexts in direct style (the Moby paper (Reppy, 2002) does so using labelled expressions), so it is *only* a matter of convenience, not feasibility.

3.4 Recursive Join Points and Fusion

We have mentioned, without stressing the point, that join points can be recursive. We have also shown that it is rather easy to identify let-bindings that can be re-expressed (more efficiently) as join points. To our complete surprise, we discovered that the combination of these two features allowed us to solve a long-standing problem with stream fusion.

[5]The parts of this sequence not specifically to do with join points were already implemented before in GHC: The Float In pass applies *float* in reverse, and the Simplifier regularly creates join points to share evaluation contexts (except that previously they were ordinary **let** bindings).

Recursive Join Points

Consider this program, which finds the first element of a list that satisfies a predicate p:

$$\begin{aligned}
&\mathit{find} = \Lambda a.\, \lambda(p : a \to \mathit{Bool})(\mathit{xs} : [a]).\\
&\quad \textbf{let}\; \mathit{go}\; \mathit{xs} = \textbf{case}\, \mathit{xs}\, \textbf{of}\\
&\qquad\qquad x : \mathit{xs}' \to \textbf{if}\, p\; x\, \textbf{then}\; \mathit{Just}\; x\\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\;\; \mathit{go}\; \mathit{xs}'\\
&\qquad\qquad [\,] \qquad\;\; \to \mathit{Nothing}\\
&\quad \textbf{in}\, \mathit{go}\; \mathit{xs}_0
\end{aligned}$$

Programmers quite often write loops like this, with a local definition for *go*, perhaps to allow *find* to be inlined at a call site. Our first observation is this: *go* is a (recursive) join point! The contification transformation of will identify *go* as a join point, and will transform the **let** to a **join**, and each call to *go* into a **jump**. Moreover, the transformed function is much more efficient because there is no longer a heap-allocated closure for *go*.

But it gets better! Because *go* is a join point, it can participate in a commuting conversion. Suppose, for example, that *find* is called from *any* like this:

$$\begin{aligned}
&\mathit{any} = \Lambda a.\, \lambda(p : a \to \mathit{Bool})(\mathit{xs} : [a]).\\
&\quad \textbf{case}\, \mathit{find}\, p\, \mathit{xs}\; \textbf{of}\; \mathit{Just}\, \_ \quad \to \mathit{True}\\
&\qquad\qquad\qquad\qquad\;\; \mathit{Nothing} \to \mathit{False}
\end{aligned}$$

The call to *find* can be inlined:

$$\begin{array}{l}
\mathit{any} = \Lambda a.\,\lambda(p : a \to \mathit{Bool})(\mathit{xs} : [a]).\\
\quad \textbf{case}\left(\begin{array}{l}
\textbf{join}\ \mathit{go}\ \mathit{xs} = \textbf{case}\ \mathit{xs}\ \textbf{of}\\
\quad x : \mathit{xs}' \to \textbf{if}\ p\,x\ \textbf{then}\ \mathit{Just}\,x\\
\qquad\qquad\qquad\quad \textbf{else}\ \ \textbf{jump}\ \mathit{go}\ \mathit{xs}'\ (\mathit{Maybe}\,a)\\
\quad [] \qquad\ \to \mathit{Nothing}\\
\textbf{in}\,\textbf{jump}\ \mathit{go}\ \mathit{xs}\ (\mathit{Maybe}\ a)
\end{array}\right)\textbf{of}\\
\quad \{\mathit{Just}\,\_ \to \mathit{True}; \mathit{Nothing} \to \mathit{False}\}
\end{array}$$

Now, we have a **case** scrutinizing a **join** so we can apply axiom *jfloat* from Figure 20. After some easy further transformations, we get

$$\begin{array}{l}
\mathit{any} = \ \Lambda a.\,\lambda(p : a \to \mathit{Bool})(\mathit{xs} : [a]).\\
\qquad \textbf{join}\ \mathit{go}\ \mathit{xs} = \textbf{case}\ \mathit{xs}\ \textbf{of}\\
\qquad\quad x : \mathit{xs}' \to \textbf{if}\ p\,x\ \textbf{then}\ \mathit{True}\\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \textbf{jump}\ \mathit{go}\ \mathit{xs}'\ \mathit{Bool}\\
\qquad\quad [] \qquad\ \to \mathit{False}\\
\qquad \textbf{in}\,\textbf{jump}\ \mathit{go}\ \mathit{xs}\ \mathit{Bool}
\end{array}$$

Look carefully at what has happened here: the consumer (*any*) of a recursive loop (*go*) has moved *all the way to the return point of the loop*, so that we were able to cancel the **case** in the consumer with the data constructor returned at the conclusion of the loop.

It turns out that this new ability to move a consumer all the way to the return points of a tail-recursive loop has direct implications for a very widely used transformation: stream fusion. The key idea of stream fusion is to represent a list (or array, or other sequence) by a pair of a *state* and a *stepper function*, thus:[6]

```
data Stream a where
  MkStream :: s -> (s -> Step s a) -> Stream a
```

There are two competing approaches to the `Step` type. In unfold/destroy fusion, first described by Svenningsson (Svenningsson, 2002), we have:

```
data Step s a = Done | Yield s a
```

Hence a stepper function takes an incoming state and either yields an element and a new state or signals the end.

Now a pipeline of list processors can be rewritten as a pipeline of stepper functions, each of which produces and consumes elements one by one. A typical stepper function for a stream transformer looks like:

```
  next s = case <incoming step> of
             Yield s’ a -> <process element>
             Done       -> <process end of stream>
```

When composed together and inlined, the stepper functions become a nest of `case`s, each scrutinizing the output of the previous stepper. It is crucial for performance that each `Yield` or `Done` expression be matched to a `case`, much as we did with `Just` and `Nothing` in the example that began Sec. 3.1. Fortunately,

[6]Note that `Stream` is an existential type, so as to abstract the internal state type `s` as an implementation detail of the stream.

case-of-case and the other commuting conversions that GHC performs are usually up to the task.

Alas, this approach requires a recursive stepper function when implementing `filter`, which must loop over incoming elements until it finds a match. This breaks up the chain of `case`s by putting a loop in the way, much as our *any* above becomes a **case** on a loop. Hence until now, recursive stepper functions have been un-fusible. Coutts et al. (Coutts et al., 2007) suggested adding a `Skip` construtor to `Step`, thus:

```
data Step s a = Done | Yield s a | Skip s
```

Now the stepper function can say to update the state and call again, obviating the need for a loop of its own. This makes `filter` fusible, but it complicates everything else! Everything gets three cases instead of two, leading to more code and more runtime tests; and functions like `zip` that consume two lists become more complicated and less efficient.

But with join points, just as with *any*, Svenningsson's original Skip-less approach fuses just fine! Result: simpler code, less of it, and faster to execute. It's a straight win.

3.5 Metatheory of F_J

Type Safety

The way to "run" a program on our abstract machine is to initialize the machine with an empty stack and an empty store. Type safety, then, says that once we start the machine, the program either runs forever or successfully returns an answer.

Theorem 2 (Type safety). *If $\varepsilon; \varepsilon \vdash e : \tau$, then either:*

1. *The initial configuration $\langle e;\ \varepsilon;\ \varepsilon\rangle$ diverges, or*

2. *$\langle e;\, \varepsilon;\, \varepsilon\rangle \mapsto^{\star} \langle A;\, \varepsilon;\, \Sigma\rangle$, for some store Σ and answer A.*

We did not give a type system for configurations, so the off-the-shelf proof of progress and preservation is not quite applicable. However, we can adapt easily enough by annotating each configuration with a well-typed term that corresponds to it. Write $\langle e/s; \Sigma; e\rangle$ (or $\langle e/c\rangle$) for an *annotated configuration.* We will need to track the connection between e and c, for which we need a few tools. Let B be a *binding context*, that is, series of **let** bindings surrounding a hole. Then write $[\![B]\!]$ for the store containing those same bindings (but with recursive groups flattened). Also, let $[\![E]\!]$ translate the evaluation context E to a stack (which is of course just another syntax for the same structure). Then let $\sim$ relate terms to configurations such that

$$B[E[e]] \sim \langle e;\, [\![E]\!];\, [\![B]\!]\rangle.$$

Finally, write

$$\langle e/c\rangle : \tau$$

when $e \sim c$ and $\varepsilon;\, \varepsilon \vdash e : \tau$, and write

$$\langle e/c\rangle \mapsto \langle e'/c'\rangle$$

if $c \mapsto c'$.

We need a few utilities before we tackle the proof.

Proposition 3. *Substitution*

1. *If $\Gamma, x : \sigma;\, \Delta \vdash e : \tau$ and $\Gamma;\, \Delta \vdash v : \sigma$, then $\Gamma, \Delta \vdash e\{x/v\} : \tau$.*

2. *If $\Gamma, a;\, \Delta \vdash e : \tau$, then $\Gamma;\, \Delta \vdash e\{a/\sigma\} : \tau\{a/\sigma\}$.*

Lemma 4. *If* $\Gamma;\, \Delta \vdash E[B[e]] : \tau$ *and variables bound by* B *aren't free in* E*, then* $\Gamma;\, \Delta \vdash B[E[\tau]]$.

Proof. By induction on E and then B. □

Now we are ready:

Lemma 5 (Progress and preservation). *If* $\langle e/c \rangle : \tau$*, then either*

1. $c \equiv \langle A;\, s;\, \Sigma \rangle$*, where* A *is an answer, or*
2. $\langle e/c \rangle \mapsto \langle e'/c' \rangle$ *for some* e' *and* c' *with* $\langle e'/c' \rangle : \tau$.

Proof. Let $c \equiv \langle e_0;\, s;\, \Sigma \rangle$; proceed by case analysis on e_0.

- For $e_0 \equiv \text{any}F[e_1]$, the *push* rule applies, and we can take $e' \equiv e$, so $e' : \tau$ holds by assumption.
- For $e_0 \equiv \textbf{let}\, vb\, \textbf{in}\, e_1$, the *bind* rule applies, and we finish with Lemma 4.
- For $e_0 \equiv \textbf{jump}\ j\ \vec{\varphi}\ \vec{v}\ \tau$, note that there must be a matching **join** in s (provable by induction on s). Then that matching **join** must have given the matching body u the same type that all of e has (by induction on s and then Σ). Thus Jump applies and we finish by Prop. 3.
- For $e_0 \equiv A$, examine s:
  - If $s \equiv \varepsilon$, we are done (case 1).
  - If $s \equiv \textbf{join}\, jb\, \textbf{in}\, s'$, then *ans* rule applies. The reduct typechecks by a standard strengthening lemma, since no label can appear free in an answer.
  - Otherwise, the outermost frame must be of the correct form according to the type of e_0, so one of β, β_τ, or *case* applies. In each case we finish with either Lemma 4 or Prop. 3.

– For $e_0 \equiv x$, we must have that $B : \Gamma$, so *bind* applies; by induction on B, we can find the proof that $u : \tau$. □

Proof (of Theorem 2). Generalize from the initial configuration to any $\langle e/c\rangle : \tau$, since clearly $e \sim \langle e;\, \varepsilon;\, \varepsilon\rangle$ and hence $\langle e/\varepsilon; \varepsilon; e\rangle : \tau$. Proceed by coinduction. By Lemma 5, either c is an answer configuration (proving case 2) or $\langle e/c\rangle \mapsto \langle e'/c'\rangle$ where $\langle e'/c'\rangle : \tau$. This may proceed forever, proving case 1, or else eventually there must be an answer. □

Correctness of the Optimization Rules

To establish the correctness of our rewriting axioms, we first define a notion of *observational equivalence.*

Definition 6. *Two terms e and e' are* observationally equivalent*, written $e \cong e'$, if, given any stack s and store Σ, either*

- *both $\langle e;\, s;\, \Sigma\rangle$ and $\langle e';\, s;\, \Sigma\rangle$ diverge, or*
- *for some Σ'_1, A_1, Σ'_2, and A_2, $\langle e;\, s;\, \Sigma\rangle \mapsto^\star \langle A_1;\, \varepsilon;\, \Sigma'_1\rangle$ and $\langle e';\, s;\, \Sigma\rangle \mapsto^\star \langle A_2;\, \varepsilon;\, \Sigma'_2\rangle$.*

The equational theory is sound with respect to observational equivalence:

Proposition 7. *If $e = e'$, then $e \cong e'$.*

Equivalence to System F

The best way to be sure that F_J can be implemented without any headaches is to show that it is equivalent to GHC's existing System F-based language. This would suggest that the introduction of join points does not allow us to write any *new* programs, only to implement existing programs more efficiently. To prove the equivalence, we establish an *erasure* procedure that removes all join points from an F_J term, leaving an equivalent System F term.

To erase the join points, we want to apply the *contify* axiom (or its recursive variant) from right to left. However, we cannot necessarily do so immediately for each join point, since *contify* only applies when all invocations are in tail position. For example, we cannot de-contify j here:

$$\mathbf{join}\, j\ x = x + 1\, \mathbf{in}\, (\mathbf{jump}\ j\ 1\ (\mathit{Int} \to \mathit{Int}))\, 2$$

Simply rewriting the join point as a function and the jump as a function call would change the meaning of the program—in fact, it would not even be well-typed:

$$\mathbf{let}\, f = \lambda x.\, x + 1\, \mathbf{in}\, f\, 1\, 2$$

However, if we apply *abort* first:

$$\mathbf{join}\, j\ x = x + 1\, \mathbf{in}\, \mathbf{jump}\ j\ 1\ \mathit{Int}$$

Now the jump is a tail call, so *contify* applies.

The *abort* axiom is not enough on its own, since the jump may be buried inside a tail context:

$$\mathbf{join}\, j\ x = x + 1\, \mathbf{in} \left(\begin{array}{l} \mathbf{case}\, b\, \mathbf{of} \\ \quad \mathit{True} \to \mathbf{jump}\ j\ 1\ (\mathit{Int} \to \mathit{Int}) \\ \quad \mathit{False} \to \mathbf{jump}\ j\ 3\ (\mathit{Int} \to \mathit{Int}) \end{array} \right) 2$$

However, this can be handled by a commuting conversion:

$$\begin{array}{l} \mathbf{join}\, j\ x = x + 1\, \mathbf{in}\, \mathbf{case}\, b\, \mathbf{of} \\ \qquad\qquad \mathit{True} \to (\mathbf{jump}\ j\ 1\ (\mathit{Int} \to \mathit{Int}))\, 2 \\ \qquad\qquad \mathit{False} \to (\mathbf{jump}\ j\ 3\ (\mathit{Int} \to \mathit{Int}))\, 2 \end{array}$$

And now *abort* applies twice and j can be de-contified.

Lemma 8. *For any well-typed term e, there is an e' such that $e' = e$ and every jump in e' is in tail position.*

By "tail position," we mean one of the holes in a tail context that starts with the binding for the join point being called. In other words, given a term

$$\mathbf{join}\, j\ \vec{a}\ \vec{x} = u\, \mathbf{in}\, L[\vec{e}],$$

the terms $\vec{e}$ are in tail position for j.

The proof of Lemma 8 relies on the observation that the places in a term that may contain free occurrences of labels are precisely those appearing in the hole of either an evaluation or a tail context. For example, the CASE typing rule propagates Δ into both the scrutinee and the branches; note that $\mathbf{case}\,\square\,\mathbf{of}\;\overrightarrow{alt}$ is an evaluation context and $\mathbf{case}\, e\, \mathbf{of}\; \overrightarrow{p \to \square}$ is a tail context. But $e\ \square$ is (in call-by-name) neither an evaluation context *nor* a tail context, and APP does *not* propagate Δ into the argument.

Thus *any* expression can be written as:

$$L[\overrightarrow{E[L'[\overrightarrow{E'[\ldots[L^{(n)}[\overrightarrow{E^{(n)}[e]}]]\ldots]}]]}], \tag{3.1}$$

which is to say a tree of tail contexts alternating with evaluation contexts, where all free occurrences of join points are at the leaves. By iterating *commute* and *abort*, we can flatten the tree, rewriting (3.1) to say that any expression can be written $L[\vec{e}]$, where each e_i is a leaf from the tree in (3.1). Hence no e_i can be expressed as $E[L[\ldots]]$ for nontrivial, non-binding[7] E and nontrivial L, and every jump to a free occurrence of a label is some e_i. Let us say a term in the above form is in *commuting-normal form*[8]. By *commute* and *abort*, every

[7]A **join** can be treated as either an evaluation context or a tail context; using *commute* to push a **join** inward is not necessarily helpful, however.

term has a commuting-normal form, and by construction, every jump in a commuting-normal form is a tail call. Thus *every label can be decontified*, and we have:

Theorem 9 (Erasure). *For any closed, well-typed* F_J *term* e*, there is a System F term* e' *such that* $e' = e$*.*

3.6 Join Points in Practice

Is is one thing to define a calculus, but quite another to use it in a full-scale optimising compiler. In this section we report on our experience of doing so in GHC.

Implementing Join Points in GHC

We have implemented System F_J as an extension to the Core language in GHC. Rather than adding two new data constructors for **join** and **jump** to the Core data type, we instead re-use ordinary **let**-bindings and function applications, distinguishing join points only by a flag *on the identifier itself*.

Thus, with no code changes, GHC treats join-point identifiers identically to other identifiers, and join-point bindings identically to ordinary **let** bindings. This is extremely convenient in practice. For example, all the code that deals with dropping dead bindings, inlining a binding that occurs just once, inlining a binding whose right-hand side is small, and so on, all works automatically for join points too.

With the modified Core language in hand, we had three tasks. First, GHC has an internal typechecker, called Core Lint, that (optionally) checks the type-correctness of the intermediate program after each pass. We augmented Core Lint for F_J according to the rules of Fig. 18.

[8]ANF is simply commuting-normal form with named intermediate values.

Second, we added a simple new contification analysis to identify let-bindings that can be converted into join points (see Sec. 3.3). Since the analysis is simple, we run it frequently, whenever the so-called occurrence analyzer runs.

Finally, the new Core Lint forensically identified several existing Core-to-Core passes that were "destroying" join points (see Sec.). Destroying a join point de-optimizes the program, so it is wonderful now to have a way to nail such problems at their source. Moreover, once Lint flagged a problem, it was never difficult to alter the Core-to-Core transformation to make it preserve join points. Here are some of the specifics about particular passes:

The Simplifier is a sort of partial evaluator responsible for many local transformations, including commuting conversions and inlining (Peyton Jones & Santos, 1998). The Simplifier is implemented as a tail-recursive traversal that builds up a representation of the evaluation context as it goes; as such, implementing the *jfloat* and *abort* axioms (Sec.) requires only two new behaviors:

- (*jfloat*) When traversing a join-point binding, copy the evaluation context into the right-hand side.
- (*abort*) When traversing a jump, throw away the evaluation context.

The Float Out pass moves **let** bindings outwards (Peyton Jones, Partain, & Santos, 1996). Moving a **join** binding outwards, however, risks destroying the join point, so we modified Float Out to leave **join** bindings alone in most cases.

The Float In pass moves **let** bindings inwards. It too can destroy join points by un-saturating them. For example, given `let j x y = ... in j 1 2`, the Float In pass wants to narrow j's scope as much as possible: `(let j x y = ... in j) 1 2`. We modified Float In so that it never un-saturates a join point.

Strictness analysis is as useful for join points as it is for ordinary **let** bindings, so it is convenient that **join** bindings are, by default, treated identically to ordinary **let** bindings. In GHC, the results of strictness analysis are exploited by the so-called worker/wrapper transform (Peyton Jones & Santos, 1998; Gill & Hutton, 2009). We needed to modify this transform so that the generated worker and wrapper are both join points. We found that GHC's *constructed product result* (CPR) analysis (Baker-Finch, Glynn, & Peyton Jones, 2004) caused the wrapper to invoke the worker inside a **case** expression, thus preventing the worker from being a join point. We simply disable CPR analysis for join points; it turns out that the commuting conversions for join points do a better job anyway.

Benchmarks

The reason for adding join points is to improve performance; expressiveness is unchanged (Sec. 3.5). So does performance improve? Table 1 presents benchmark data on allocations, collected from the standard `spectral`, `real` and `shootout` NoFib benchmark suites[9]. We ran the tests on our modified GHC branch, and compared them to the GHC baseline to which our modifications were applied. Remember, the baseline compiler *already* recognises join points in the back end and compiles them efficiently (Sec.); the performance changes here come from preserving and exploiting join points during optimization.

We report only heap allocations because they are a repeatable proxy for runtime; the latter is much harder to measure reliably. All tests omitted from the tables had an improvement in allocations, but less than 0.3%.

[9]The `imaginary` suite had no interesting cases. We believe this is because join points tend to show up only in fairly large functions, and the `imaginary` tests are all micro-benchmarks.

`spectral`

Program	Allocs
fibheaps	-1.1%
ida	-1.4%
nucleic2	+0.2%
para	-4.3%
primetest	-3.6%
simple	-0.9%
solid	-8.4%
sphere	-3.3%
transform	+1.1%
(45 others)	
Min	-8.4%
Max	+1.1%
Geo. Mean	-0.4%

`real`

Program	Allocs
anna	+0.5%
cacheprof	-0.5%
fem	+3.6%
gamteb	-1.4%
hpg	-2.1%
parser	+1.2%
rsa	-4.7%
(18 others)	
Min	-4.7%
Max	+3.6%
Geo. Mean	-0.2%

`shootout`

Program	Allocs
k-nucleotide	-85.9%
n-body	-100.0%
spectral-norm	-0.8%
(5 others)	
Min	-100.0%
Max	+0.0%
Geo. Mean	n/a

TABLE 1. Benchmarks from the `spectral`, `real`, and `shootout` NoFib suites.

There are some startling figures: using join points eliminated *all* allocations in `n-body` and 85.9% in `k-nucleotide`. We caution that these are highly atypical programs, already hand-crafted to run fast. Still, it seems that our work may make it easier for performance-hungry authors to squeeze more performance out of their inner loops.

The complex interaction between inlining and other transformations makes it impossible to give *guaranteed* improvements. For example, improving a function f might make it small enough to inline into g, but this may cause g to become too *large* to inline elsewhere, and that in turn may lose the optimization opportunities previously exposed by inlining g. GHC's approach is heuristic, aiming to make losses unlikely, but they do occur, including a 1.1% increase in allocations in `spectral/transform` and a 3.6% increase in `real/fem`.

Beyond Benchmarks

These benchmarks show modest but fairly consistent improvements for existing, unmodified programs. But we believe that the systematic addition of join points may have a more significant effect on programming patterns. Our discussion of fusion in Sec. 3.4 is a case in point: with join points we can use skip-less unfoldr/destroy streams without sacrificing fusion. That knowledge in turn affects the way in which libraries are written: they can be smaller and faster.

Moreover, the transformation pipeline becomes more robust. In GHC today, if a "join point" is inlined we get good fusion behavior, but if its size grows to exceed the (arbitrary) inlining threshold, suddenly behavior becomes much worse. An innocuous change in the source program can lead to a big change in execution time. That step-change problem disappears when we formally add join points.

CHAPTER IV

Lazy Functions as Processes

Much of the text in this chapter comes from (Downen, Maurer, Ariola, & Varacca, 2014), which was a collaboration with Paul Downen (UO), Zena M. Ariola (UO), and Daniele Varacca (UPD). I was the primary designer of the new language.

In Chapter III, we saw that we could improve the results of GHC's optimizer by giving it more opportunities for code motion. However, code motion is inherently limited by the level of abstraction presented by the language. By design, Core leaves implicit the operations that make lazy evaluation efficient, namely the update to each memo-thunk once its value has been computed. Conceivably, being able to move these updates around might prove beneficial. For instance, suppose we have $x = \mathit{fst}\ y$ and y is demanded before x. As soon as y is updated, x becomes such a quick operation that we might as well update it right away, thus avoiding an indirect jump the first time x is accessed. But in order to group together updates in this way, we would first need to extend Core to make the memo-thunk updates explicit.

While actually implementing this in GHC remains future work, this chapter demonstrates one possible approach by showing how to take a fragment of a language where the update is easily expressed and translate it into a more practical form for implementation in an optimizing compiler.

Continuations and continuation-passing style (CPS) provide powerful and versatile tools for understanding programming languages (Reynolds, 1993). By representing the "future of the program" as a first-class entity, a *CPS transform* gives a denotational semantics for a programming language in terms of a simple, well-understood low-level language. In a particularly influential use of continuations, Plotkin (Plotkin, 1975) demonstrated how a CPS transform can weave an implementation strategy for a program into the syntax of the program itself. This methodology gave rise to the call-by-value λ-calculus and was instrumental in closing the gap between the theory and practice of functional languages.

Since that time, CPS transforms have continued to further our understanding of programming languages. The call-by-value CPS transform was more descriptive than Plotkin's original call-by-value λ-calculus, motivating a more thorough study of strict functional languages; in turn, this lead to more advanced techniques for reasoning about programs in continuation-passing style and to a more complete development of the call-by-value λ-calculus (Sabry & Felleisen, 1993; Sabry & Wadler, 1997). CPS transforms and related techniques have also provided a formal method for reasoning about effects, such as mutable references and non-local jumps, that lie outside of the pure model of the λ-calculus. Of particular note is delimited control, especially the **shift** and **reset** operators (Danvy & Filinski, 1989), which were originally developed by defining them in continuation-passing style.

Flexible as CPS transforms are, they inherit some of the limitations of the λ-calculus. A λ-term describes a *sequential* and (due to confluence) *determinate* computation. Features such as parallelism, distributed computation, and nondeterminacy have no natural expression in the base λ-calculus. We can extend the λ-calculus with such features, but we argue that it would be better to find a different target language altogether, one with these features "out-of-the-box."

A good candidate for such a target language is the π-calculus (Milner, Parrow, & Walker, 1992; Sangiorgi & Walker, 2003), a process calculus describing interacting systems running in parallel. Early in the study of the π-

calculus, Milner proved (Milner, 1992) that the π-calculus was powerful enough to embed the λ-calculus by a simple interpretation function. His π-encoding was formulated from scratch, but later Sangiorgi (Sangiorgi, 1999) discovered that we could also derive a π-encoding from a CPS transform, if we can account for the first-order nature of the π-calculus—processes don't transmit processes to each other the way higher-order functions pass functions as arguments. As an alternative, Amadio (Amadio, 2011) showed that we could instead translate a CPS term into a language of first-order functions and bound names, which then corresponds directly to the π-calculus.

This analysis has been applied to CPS transforms for both call-by-name and call-by-value evaluation. In this paper, we extend it to two calculi modelling real-world implementations: first, a variation on call-by-value that reflects the way a typical interpreter performs variable lookup; and second, the *call-by-need λ-calculus* (Ariola et al., 1995; Ariola & Felleisen, 1997), which models implementations of call-by-name languages that cache each value computed.

We start, in Section 4.1, with an introduction of the call-by-name and call-by-value lambda-calculi. We present their operational semantics and a uniform CPS transform, from which we derive CBN and CBV π-encodings, along with abstract machines. In Section 4.5, we consider call-by-need evaluation; we present a novel call-by-need CPS transform, which leads to an interesting concept in its own right: the notion of *constructive update*. From the new CPS transform, we derive the call-by-need π-encoding and an abstract machine in much the same way as we did for CBN and CBV.

4.1 Call-by-Name and Call-by-Value

The λ-calculus, defined by Church (Church, 1932) in the 1930s, is a simple yet powerful model of computation. It consists of only three parts: *functions* from inputs to outputs, *variables* that stand for the inputs, and *applications* that invoke the functions:

$$\text{Terms:} \quad M, N ::= \lambda x.\, M \mid x \mid MN$$

As the λ-calculus can be a foundation of both strict and lazy languages, a λ-term M can be evaluated according to different *evaluation strategies,* which dictate the operation to be performed first. The two most studied evaluation strategies are *call-by-name* (CBN) and *call-by-value* (CBV). In CBN evaluation, the argument to a function is kept unevaluated as long as possible, then evaluated each time its value is required for computation to continue. In CBV evaluation, the argument is always evaluated before the function receives it. It is usually better to precompute arguments, as CBV does, since then an argument will not be evaluated more than once; however, if the function does not actually use the value, computing it is wasteful. In the extreme case, if the argument *diverges* (that is, loops forever) but is never used by the function, CBV diverges when CBN does not.

The distinct CBN and CBV reduction strategies are captured by the two reduction rules, β_n and β_v (see Figs. 22 and 23). A β_n-reduction starts from a term $(\lambda x.\, M)N$, then proceeds by substituting the unevaluated argument N into the body wherever x appears. Thus the function call precedes the evaluation of the argument. In contrast, a β_v-reduction evaluates the argument first: Only a *value* may be substituted into the body. A value is either a variable or a function literal (also called a λ-*abstraction*).

$$\text{Evaluation Contexts:}\quad E ::= \Box \mid EM$$

$$(\lambda x.\, M)N \to M\{N/x\} \qquad\qquad \beta_n$$

FIGURE 22. The call-by-name λ-calculus, λ_n.

$$\begin{aligned} \text{Values:}\quad & V ::= x \mid \lambda x.\, M \\ \text{Evaluation Contexts:}\quad & E ::= \Box \mid EM \mid VE \end{aligned}$$

$$(\lambda x.\, M)V \to M\{V/x\} \qquad\qquad \beta_v$$

FIGURE 23. The call-by-value λ-calculus, λ_v.

A term that pattern-matches the left-hand side of a reduction rule is called a *redex*, short for *reducible expression.* For instance, $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$ is a call-by-name redex. However, it is not a call-by-value redex, because the argument $(\lambda y.\, y)(\lambda z.\, z)$ is not a value. If a redex somewhere in M is reduced, and the resulting term is N, we write $M \to N$ and say that M *reduces to* N; we also write $\to^\star$ for zero or more reductions and $\to^+$ for one or more reductions.

To complete the semantics, one has to specify where a reduction should take place. Felleisen and Friedman (Felleisen & Friedman, 1986) introduced a concise way to do so, using *evaluation contexts.* A context is a "term with a hole": It is the outer portion of some term, surrounding a single occurrence of the symbol $\Box$. A language's evaluation contexts delineate the places in a term where evaluation may take place. For instance, consider the grammar for evaluation contexts in Fig. 22: An evaluation context can be either just a hole (the *trivial* or *top-level* context, $\Box$) or EM, a subcontext applied to an argument. Thus CBN evaluation can take place either at the top level or within the operator in a function application, but not within the argument (since arguments are left unevaluated). The CBV calculus also has contexts VE, so once the function in an application has become a value, CBV evaluation proceeds within the argument.

Given an evaluation context E, we write $E[M]$ for E with the term M in place of the hole, and we say M is *plugged into* E. (This notation applies to general contexts as well.) The inverse of the "plugging in" operation is *decomposition*, and it plays a critical role in evaluation by finding where to perform the next reduction. For example, we can decompose $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$ in CBN as $E[M]$ where E is the top-level context $\square$ and M is the entire term. In CBV, we can decompose it with E being $(\lambda x.\, xx)\square$ and M being $(\lambda y.\, y)(\lambda z.\, z)$, since the next step of CBV evaluation is to reduce the argument.

If $M \equiv E[M']$ and M' is a redex reducing to N', then we write $M \mapsto N$ where $N \triangleq E[N']$. In other words, $\mapsto$ denotes reduction only within an evaluation context, which we call *standard reduction* or *evaluation.* Often we will say that M *steps* or *takes a step* in this case. As before, we write $\mapsto^\star$ for the reflexive and transitive closure and $\mapsto^+$ for the transitive closure.

Finally, we introduce notations for the possible *observations* one can make about a term. These are the potential outcomes of computation, without regard to the particular steps taken. If M is a λ-abstraction, which we also call an *answer*, we write AM. If M *evaluates* to an answer (perhaps because it is one), we write $M \Downarrow$. A term M with no possible evaluation step, but which is *not* an answer, is called *stuck*, written sM; a term M that evaluates to a stuck term *gets stuck*, written $M \not\Downarrow$. If M *never* finishes evaluating—that is, it takes infinitely many evaluation steps—it is said to *diverge*, written $M \Uparrow$.

Example 10. Consider the term $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$. CBN and CBV evaluate the term differently (the redex at each step is shaded):

$$\begin{aligned}
&(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))\\
&\quad\mapsto ((\lambda y.\, y)(\lambda z.\, z))((\lambda y.\, y)(\lambda z.\, z))\\
&\quad\mapsto (\lambda z.\, z)((\lambda y.\, y)(\lambda z.\, z))\\
&\quad\mapsto (\lambda y.\, y)(\lambda z.\, z)\\
&\quad\mapsto (\lambda z.\, z)
\end{aligned}$$

$$\begin{aligned}
&(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))\\
&\quad\mapsto (\lambda x.\, xx)(\lambda z.\, z)\\
&\quad\mapsto (\lambda z.\, z)(\lambda z.\, z)\\
&\quad\mapsto (\lambda z.\, z)
\end{aligned}$$

CBN reduces the outer β-redex immediately, substituting the argument as is. This duplicates work, since the β_n-redex $(\lambda y.\, y)(\lambda z.\, z)$ now appears for each x in the body of $\lambda x.\, xx$. Instead, CBV evaluates the argument first, reducing it to a value before substituting, thus saving one reduction.

4.2 A Uniform CPS Transform

As an alternative to specifying the semantics of a language in terms of rewrite rules for programs, one can specify a function that "compiles," or transforms, programs into some lower-level form. The advantage is that analyzing a lower-level form is easier, since the syntax itself prescribes how a program should be executed, just as assembly code specifies not only calculations but which registers, including the program counter, to use to perform them. A transform into *continuation-passing style*, called a *CPS transform*, is an example of such a compilation function. It produces λ-terms whose evaluation order is predetermined: The same calculations will be performed, in the same order, by call-by-name or call-by-value evaluation. The

$$\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, k(\lambda(x,k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \begin{cases} \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![N]\!]k', k)) & \text{CBN} \\ \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \mathcal{C}[\![N]\!](\lambda w.\, v(\lambda k'.\, k'w, k))) & \text{CBV} \end{cases}
\end{aligned}$$

FIGURE 24. A uniform CPS transform for call-by-name and call-by-value.

trick is to pass only precomputed values as arguments to functions, making the question of when to evaluate arguments moot. Then, rather than returning its result in the usual way, a CPS function passes the result to one of its arguments, the so-called *continuation.* A continuation represents the evaluation context in which a function was invoked; hence it plays a similar role to the *call stack* used in most computer architectures. Since their evaluation contexts differ, we can elucidate the difference between CBN and CBV evaluation by translating each to continuation-passing style.

We focus on a *uniform* CPS transform $\mathcal{C}$, given in Fig. 24, so called because the translations for variables and abstractions are the same between CBN and CBV. This uniformity highlights the differences in evaluation order by varying only the translation of applications. Specifically, once M has evaluated to a function v, the continuation in the CBN transform invokes v immediately, passing it the unevaluated CPS term $\lambda k'.\, \mathcal{C}[\![N]\!]k'$ as x. Evaluating a variable is done by invoking it with a continuation, so each invocation of x within the body of v will evaluate the argument. The CBV transform evaluates M the same way, but its continuation does not use v immediately; instead, it evaluates N to a function w, and only *its* continuation invokes v. This time, the x argument is a function that immediately passes w to the continuation; therefore each invocation of x within the body of v immediately returns the precomputed argument value w.

Example 11. Consider the term $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$ from above, calling it M for now. In CBN (the redex is always the whole CPS term, so we omit the shading):

$$\begin{aligned}
\mathcal{C}[\![M]\!]k &\triangleq \mathcal{C}[\![\lambda x.\, xx]\!](\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k)) \\
&\triangleq (\lambda k.\, k(\lambda(x, k').\, \mathcal{C}[\![xx]\!]k'))(\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k)) \\
&\mapsto (\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]k', k)(\lambda(x, k').\, \mathcal{C}[\![xx]\!]k')
\end{aligned}$$

This last reduction step duplicates work, as the evaluation of $\mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!]$ must now occur twice.

Now for CBV:

$$\begin{aligned}
\mathcal{C}[\![M]\!]k &\triangleq \mathcal{C}[\![\lambda x.\, xx]\!](\lambda v.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!](\lambda w.\, v(\lambda k.\, kw, k))) \\
&\triangleq (\lambda k.\, k(\lambda(x, k').\, \mathcal{C}[\![xx]\!]k'))(\lambda v.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!](\lambda w.\, v(\lambda k.\, kw, k))) \\
&\mapsto (\lambda v.\, \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!](\lambda w.\, v(\lambda k.\, kw, k)))(\lambda(x, k').\, \mathcal{C}[\![xx]\!]k')
\end{aligned}$$

The function has evaluated to v, but this time we evaluate the argument next:

$$\mapsto \mathcal{C}[\![(\lambda y.\, y)(\lambda z.\, z)]\!](\lambda w.\, (\lambda(x, k').\, \mathcal{C}[\![xx]\!]k')(\lambda k.\, kw, k))$$

Once the argument is computed as w, *then* the function will be invoked, but this time with x being a function that immediately passes along the precomputed value w.

The uniform CPS transform reflects the behavior of common language implementations: Evaluation always stops at a λ, and a variable always causes a lookup (hence a free variable halts execution). However, these behaviors don't faithfully represent the full theory of the λ-calculus. For instance, the calculus is often considered with the η rule in addition to β. An η-reduction takes $\lambda x.\, Mx$ to just M whenever x does not appear in M. For a free variable y, then, the term $\lambda x.\, yx$ would reduce to y and then become stuck, whereas the uniform CPS transform gives a term that immediately returns the value $\lambda x.\, yx$ rather than becoming stuck.

The η rule is often considered unimportant for language implementations: Nearly all compilers and interpreters for functional languages stop evaluating when they find a λ. In fact, Plotkin's CBN CPS transform (Plotkin, 1975), which is very similar to the CBN fragment of our uniform transform, does not validate η either. The CBV fragment, however, differs more fundamentally: It doesn't follow the conventional β_v rule, either. In the CBV λ-calculus, a variable is considered a value, yet few compilers or interpreters operate this way: The term $(\lambda x.\, \lambda y.\, y)z$ should reduce to $\lambda y.\, y$, but a typical implementation would attempt to evaluate z and raise an "unbound variable" error. Accordingly, the CBV portion of the uniform CPS transform produces a term that becomes stuck on z rather than reducing to a value.

Therefore the CBV language truly implemented by the uniform CPS transform is not the one given in Fig. 23. Rather, it implements a calculus that further restricts the β_v rule to apply only to a λ-abstraction as an argument. Equivalently, this revised calculus consideres only a λ-abstraction to be a value. From now on, then, when we speak of the call-by-value calculus, we will refer to the version in Fig. 25. In particular, $\mapsto$ will refer to the restricted notion of evaluation context.

In most cases, our departure from orthodoxy will make no difference. In standard reductions of closed terms, it never happens that a free variable

$$\begin{aligned} \text{Values:} \quad & V ::= \lambda x.\, M \\ \text{Evaluation Contexts:} \quad & E ::= \Box \mid EM \mid VE \end{aligned}$$

$$(\lambda x.\, M)V \rightarrow M\{V/x\} \qquad\qquad \beta_v$$

FIGURE 25. A revised call-by-value λ-calculus.

$$\begin{aligned} \text{Terms:} \quad & M, N ::= V(W^+) \\ \text{Values:} \quad & V, W ::= x \mid \mathbf{ret} \mid \lambda(x^+).\, M \\ \text{Evaluation Contexts:} \quad & E ::= \Box \end{aligned}$$

$$(\lambda(x^+).\, M)(V^+) \rightarrow M\{V^+/x^+\} \qquad\qquad \beta$$

FIGURE 26. The syntax and semantics of the CPS λ-calculus, λ_{cps}.

appears as an argument, and thus it does not matter whether we consider it a value or not.

The CPS Language λ_{cps}

The terms produced by the uniform CPS transform comprise a restricted λ-calculus. The grammar is given in Fig. 26. In an application, the function must be a value, and it can take one or two arguments, which must also be values; we denote this $V(W^+)$ (we will omit parentheses when there is one argument). A value is a variable, a λ-abstraction, or the constant **ret**. Note that the body of an abstraction must again be a CPS term—that is, an application. In CPS, a function never returns to its caller; it only performs more function calls. Accordingly, the only evaluation context is the trivial context $\Box$, as the redex is always at the top level.

There are three kinds of value that appear in a CPS term:

Thunks A *thunk* is a suspended computation. In CPS terms, this is a function $\lambda k.\, M$ that takes a continuation, calculates a result, then passes the result to the continuation. Each term of the form $\mathcal{C}[\![M]\!]$ is a thunk. In the uniform CPS transform, variables are represented by thunks.

Continuations A *continuation* is a handler for a result; it has the form $\lambda v.\, M$. It takes a computed source value and performs the next step of evaluation. We can see it as a reification of a term's evaluation context from the source language.

Source Values Each value from the source calculus has a CPS encoding. As we are translating from calculi having only functions as values, we need only consider how to encode a function. Namely, a source function becomes a binary function $\lambda(x, k).\, M$ that takes a thunk x for computing the argument and a continuation k to invoke with the result.

Before we consider observations, we should consider what it means for a CPS program to be evaluated. A term $\mathcal{C}[\![M]\!]$ is an inert λ-abstraction; it must be given a continuation as its argument for evaluation to occur. This argument represents the context in which to evaluate M. If we consider M to be the whole program, we need an *initial continuation* to represent the top-level context. Thus we introduce the constant **ret**; to evaluate M as a CPS program, then, one writes $\mathcal{C}[\![M]\!]\,\mathbf{ret}$. If one thinks of a term $\mathcal{C}[\![M]\!]$ as meaning "evaluate M and then," $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ then reads "evaluate M and then return." Thus **ret** is analogous to the C function `exit`, which terminates execution and yields its argument as the result of the program.

Since an answer is the result of successful computation, then, a CPS answer is a term of the form $\mathbf{ret}\, V$, for a λ-abstraction V.[1] Thus AM means that M has the form $\mathbf{ret}(\lambda(x^+).\, N)$, and $M \Downarrow$ means that M evaluates to such a term. As before, sM means that M is stuck (hence not an answer); $M \not\Downarrow$ means that M gets stuck; and $M \Uparrow$ means that M diverges.

Environment-Based CPS Transform

So far, we have expressed argument passing by *substitution*: Each β-reduction substitutes the arguments for the free occurences of the corresponding variables. Effectively, the term is rewritten, with a copy of the argument in place of each occurrence. Interpreters typically operate differently: Each argument is put into an *environment*, indexed by the variable it is bound to. Then, when a variable appears as a function being invoked, its value is retrieved from the environment.

We can simulate this mechanism by giving a *name* to each abstraction in argument position, substituting only names during β-reduction, and copying the value only as necessary. This is analogous to graph rewriting and can be captured by extending the syntax with a **let** construct (Ariola & Klop, 1996): A bound name identifies a node in a graph. However, we prefer an alternative syntax which expresses the dynamic allocation of names. We write $\nu x.\, x := \lambda(x^{+}).\, M$ **in** N to indicate that a new name x is generated and a λ-abstraction is bound to it. Note that we will always bind an abstraction to a name immediately after allocation and only then. The value-named CPS λ-calculus, $\lambda_{cps,vn}$, is given in Fig. 27. Each term is now an application inside some number of bindings, which effectively serve as the environment. Each argument to an application must be a variable.

Note that we now have nontrivial evaluation contexts, unlike with λ_{cps}, whose only evaluation context was $\Box$. However, the contexts in $\lambda_{cps,vn}$ do not

[1]In principle, we could avoid adding a constant by simply using some free variable k for the initial continuation. We would then have to have a predicate $\Downarrow_k$ for each name k, and correctness theorems would be quantified over k. Thus **ret** is merely a convenience.

$$\begin{array}{rrl}
\text{Terms:} & M, N ::= & V(x^+) \\
& & \mid \nu x.\, x := \lambda(x^+).\, M \ \mathbf{in}\ N \\
\text{Values:} & V ::= & x \mid \mathbf{ret} \mid \lambda(x^+).\, M \\
\text{Binding Contexts:} & B ::= & [] \mid \nu x.\, x := \lambda(x^+).\, M \ \mathbf{in}\ B \\
\text{Eval. Contexts:} & E ::= & B
\end{array}$$

$$\begin{array}{rcll}
(\lambda(x^+).\, M)(y^+) & \to & M\{y^+/x^+\} & \beta \\
\begin{array}{c}\nu f.\, f := \lambda(x^+).\, M \\ \mathbf{in}\ E[f(y^+)]\end{array} & \to & \begin{array}{c}\nu f.\, f := \lambda(x^+).\, M \\ \mathbf{in}\ E[(\lambda(x^+).\, M)(y^+)]\end{array} & \mathit{deref}
\end{array}$$

FIGURE 27. The value-named CPS λ-calculus, $\lambda_{cps,vn}$.

specify work to be done but simply bindings for variables. To emphasize this, we call a context providing only bindings a *binding context*, and say that a CPS calculus has only binding contexts as evaluation contexts.

To convert an unnamed term to a named term, we introduce a *naming transform*, $\mathcal{N}$. The naming transform goes through all arguments appearing in a term, moving each λ-abstraction into a new variable.

$$\begin{aligned}
\mathcal{N}[\![V(\lambda(x^+).\, M)]\!] &\triangleq \nu y.\, y := \lambda(x^+).\, \mathcal{N}[\![M]\!]\ \mathbf{in}\ \mathcal{N}[\![V(y)]\!] \\
\mathcal{N}[\![V(\lambda(x^+).\, M, W)]\!] &\triangleq \nu y.\, y := \lambda(x^+).\, \mathcal{N}[\![M]\!]\ \mathbf{in}\ \mathcal{N}[\![V(y, W)]\!] \\
\mathcal{N}[\![V(y, \lambda(x^+).\, M)]\!] &\triangleq \nu z.\, z := \lambda(x^+).\, \mathcal{N}[\![M]\!]\ \mathbf{in}\ \mathcal{N}[\![V(y, z)]\!] \\
\mathcal{N}[\![(\lambda(x^+).\, M)(y^+)]\!] &\triangleq (\lambda(x^+).\, \mathcal{N}[\![M]\!])(y^+) \\
\mathcal{N}[\![f(y^+)]\!] &\triangleq f(y^+)
\end{aligned}$$

For clarity, here we assume that each function has at most two arguments, as is true for our CPS terms; $\mathcal{N}$ generalizes straightforwardly by iteration.

The uniform CPS transform under the naming transform is given in Fig. 28.

$$
\begin{aligned}
\mathcal{C}_{vn}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}_{vn}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, \nu f.\, f \coloneqq \lambda(x,k').\, \mathcal{C}_{vn}[\![M]\!]k' \mathbf{\,in\,} kf \\
\mathcal{C}_{vn}[\![MN]\!] &\triangleq \begin{cases}
\lambda k.\, \nu k'.\, k' \coloneqq \big(\lambda v. & \\
\quad \nu x.\, x \coloneqq \lambda k''.\, \mathcal{C}_{vn}[\![N]\!]k'' & \text{CBN} \\
\quad \mathbf{in\,} v(x,k)\big) \mathbf{\,in\,} \mathcal{C}_{vn}[\![M]\!]k' & \\
\lambda k.\, \nu k'.\, k' \coloneqq \big(\lambda v.\, \nu k''.\, k'' \coloneqq (\lambda w. & \\
\quad \nu x.\, x \coloneqq \lambda k'.\, k'w \mathbf{\,in\,} v(x,k)) & \text{CBV} \\
\quad \mathbf{in\,} \mathcal{C}_{vn}[\![N]\!]k''\big) \mathbf{\,in\,} \mathcal{C}_{vn}[\![M]\!]k' &
\end{cases}
\end{aligned}
$$

FIGURE 28. Uniform CPS transform in named form.

Proposition 12. $\mathcal{C}_{vn}[\![M]\!] \equiv \mathcal{N}[\![\mathcal{C}[\![M]\!]]\!]$.

Proof. Straightforward induction on M. □

4.3 Preservation of Observations

We show correctness of the value-named uniform CPS transform in two steps. We start with the correctness of the unnamed transform, then prove the correctness of the naming step.

Proof Methodology

For a CPS transform to be considered correct, we would want it to preserve termination (Meyer & Cosmadakis, 1988):

Criterion 13. $M \Downarrow$ *iff* $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$.

In order to prove Criterion 13, we want to proceed by induction on the evaluation steps. However, in order for the induction to go through, we need to establish an *invariant*: Something that is true at the beginning of evaluation and remains true after each step. For $\mathcal{C}$, the simplest invariant one can imagine would be this:

$$\begin{tikzcd}
M \arrow[r, mapsto] \arrow[d] & N \arrow[d, dashed] \\
\mathcal{C}\llbracket M \rrbracket K \arrow[r, mapsto, dashed] & \mathcal{C}\llbracket N \rrbracket K
\end{tikzcd} \tag{4.1}$$

In words, for any continuation K, whenever M reduces to N, $\mathcal{C}\llbracket M \rrbracket K$ reduces to $\mathcal{C}\llbracket N \rrbracket K$.[2] However, this invariant does not hold. One reason is that the CPS transform introduces many *administrative redexes* into the term. These are intermediate computations that do not correspond to actual β-reductions in the source language. (Non-administrative redexes are called *proper*.) Hence one step for M may correspond to many in $\mathcal{C}\llbracket M \rrbracket K$. Thus consider:

$$\begin{tikzcd}
M \arrow[r, mapsto] \arrow[d] & N \arrow[d, dashed] \\
\mathcal{C}\llbracket M \rrbracket K \arrow[r, mapsto, dashed] & \mathcal{C}\llbracket N \rrbracket K
\end{tikzcd} \tag{4.2}$$

Unfortunately, there is a more serious issue with (4.2). As noted by Plotkin (Plotkin, 1975), administrative reductions do not line up with the CPS transform in this way. Because $\mathcal{C}\llbracket N \rrbracket K$ introduces administrative redexes of its own, the true situation is this:

$$\begin{tikzcd}
M \arrow[rr, mapsto] \arrow[d] & & N \arrow[d, dashed] \\
\mathcal{C}\llbracket M \rrbracket K \arrow[r, mapsto, dashed] & P & \mathcal{C}\llbracket N \rrbracket K \arrow[l, mapsto, dashed]
\end{tikzcd} \tag{4.3}$$

Plotkin's solution was to derive a new transform that eliminated these initial administrative redexes, thus regaining (4.2). The problem with this and similar solutions (Danvy & Filinski, 1992; Danvy & Nielsen, 2003) is that the resulting transforms are more complex and difficult to reason about than the

[2]Ultimately, of course, we observe what happens when K is **ret**. But there is nothing special about **ret**; we expect our diagrams to hold for any K.

original CPS transform. For instance, usually such administration-free CPS transforms are non-compositional (Danvy & Nielsen, 2003).

Instead of changing the transform, we can further loosen the invariant using the *bisimulation* technique. Bisimulation is an alternative approach to soundness and completeness that requires only that we find *some* suitable relation to act as the invariant. Given a relation $\sim$, we can use it to prove Criterion 13 so long as the following hold:[3]

$$\begin{tikzcd}
M \arrow[d, phantom, "\sim"] & M \arrow[r, mapsto] \arrow[d, phantom, "\sim"] & N \arrow[d, phantom, "\sim"] & M \arrow[r, mapsto, dashed] \arrow[d, phantom, "\sim"] & N \arrow[d, phantom, "\sim"] \\
\mathcal{C}[\![M]\!]\,\mathbf{ret} & P \arrow[r, mapsto, dashed] & Q & P \arrow[r, mapsto] & Q
\end{tikzcd}$$

$$\begin{tikzcd}
M\downarrow \arrow[d, phantom, "\sim"] & & M \arrow[r, mapsto, dashed] \arrow[d, phantom, "\sim"] & AN \\
P \arrow[r, mapsto, dashed] & AQ & Q\downarrow &
\end{tikzcd} \tag{4.4}$$

So, M is related to its image under the transform; when either related term takes a step, the other can take some number of steps to remain in the relation; and if either is an answer, the other evaluates to an answer.

Once we have that evaluation to an answer is preserved, what can we say about a stuck term? It could happen that M is stuck but $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ loops forever, or vice versa. Thus we consider an additional criterion:

Criterion 14. *$M \not\Downarrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \not\Downarrow$.*

[3]Technically, it is the second and third diagrams that characterize a bisimulation. The others are additional properties that we need in order to finish the proof. The fourth and fifth are very similar to the requirements on a *barbed bisimulation* (Milner & Sangiorgi, 1992).

To prove Criterion 14, we require two more properties of the simulation $\sim$, in addition to those in (4.4):

$$\begin{tikzcd}
M \not\downarrow & & M \arrow[r, dashed, mapsto] & sN \\
\sim & & \sim & \\
P \arrow[r, dashed, mapsto] & sQ & P \not\downarrow &
\end{tikzcd} \tag{4.5}$$

In words, if either M or P is stuck, then the other must get stuck.

The final observation that we want to preserve is divergence:

Criterion 15. *$M \Uparrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Uparrow$.*

However, because evaluation is deterministic in our calculi, we can get Criterion 15 "for free" from Criteria 13 and 14: If one term diverges, it can neither reduce to an answer nor get stuck, and hence the other term can only diverge.

We can further simplify the proof methodology thanks to Leroy's observation (Leroy, 2009) that if the source language is deterministic, the forward simulation is sufficient, so long as each source evaluation step maps to at least one CPS step.[4] In short, it will suffice to show:

$$\begin{tikzcd}
M & & M \arrow[r, mapsto] & N \\
\sim & & \sim & \sim \\
\mathcal{C}[\![M]\!]\,\mathbf{ret} & & P \arrow[r, dashed, mapsto, "+"] & Q \\
M \downarrow & & M \not\downarrow & \\
\sim & & \sim & \\
P \arrow[r, dashed, mapsto] & AQ & P \arrow[r, dashed, mapsto] & sQ
\end{tikzcd} \tag{4.6}$$

From these properties and determinacy, we can prove both directions of Criteria 13 to 15. The forward direction follows directly by induction. For the

[4]Otherwise, a diverging source term could translate to an answer.

backward direction of Criterion 13, we can argue by contraposition: If M *does not* reduce to an answer, then it must either diverge or get stuck. If it diverges, then by the second diagram in (4.6), it must hold that $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ diverges, and hence by determinacy it cannot reduce to an answer. Similarly, if M gets stuck, $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ must get stuck, and hence cannot reduce to an answer. The reasoning for the backward directions of Criteria 14 and 15 is similar.

Correctness of the CPS Transform

We define the simulation $\sim$ by comparing terms in a way that ignores all administrative reductions. We consider a λ-abstraction administrative when it always forms an administrative redex. We mark these administrative λ-abstractions by placing a line over the λ, as in $\bar{\lambda}k.\,M$. The explicitly marked uniform CPS transform is then:

$$\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \bar{\lambda}k.\,xk \\
\mathcal{C}[\![\lambda x.\,M]\!] &\triangleq \bar{\lambda}k.\,k(\lambda(x,k').\,\mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \begin{cases} \bar{\lambda}k.\,\mathcal{C}[\![M]\!](\bar{\lambda}v.\,v(\bar{\lambda}k'.\,\mathcal{C}[\![N]\!]k',k)) & \text{CBN} \\ \bar{\lambda}k.\,\mathcal{C}[\![M]\!](\bar{\lambda}v.\,\mathcal{C}[\![N]\!](\bar{\lambda}w.\,v(\bar{\lambda}k'.\,k'w,k))) & \text{CBV} \end{cases}
\end{aligned}$$

Notice that the only proper λ-abstractions are the ones that correspond to a λ-abstraction from the original term, since these are the abstractions whose reductions correspond to the actual β-reductions in the source language. To distinguish administrative computation, we introduce the reduction relation $\to_{ad}$, defined by the *administrative β-rule*:

$$(\bar{\lambda}(x^+).\,M)(V^+) \to_{ad} M\{V^+/x^+\} \qquad \beta_{ad}$$

Keeping to our notational conventions, the reflexive and transitive closure of $\rightarrow_{ad}$ is $\rightarrow^{\star}_{ad}$ and its transitive closure is $\rightarrow^{+}_{ad}$. Also, its reflexive, *symmetric*, and transitive closure is $=_{ad}$; in other words, $=_{ad}$ extends $\rightarrow^{\star}_{ad}$ by allowing rules to be applied in reverse (as *expansions* rather than reductions). Furthermore, $\mapsto_{ad}$ stands for a standard administrative reduction, which is to say an administrative reduction in the empty context (at top level), and $\mapsto^{\star}_{ad}$ are $\mapsto^{+}_{ad}$ are the usual closures. We will also use the subscript *pr* in place of *ad* to denote a proper reduction. Finally, $\mapsto^{+}_{pr1}$ is short for $\mapsto^{\star}_{ad}\mapsto_{pr}\mapsto^{\star}_{ad}$, which is to say, some number of standard reductions, exactly one of which is proper.

If a term cannot take an administrative standard reduction, then for the moment, the administrative work in that term is finished. Hence, if we consider the administrative subcalculus of λ_{cps}, such a term is the result, or *answer*, of administrative computation. Therefore let a term with no administrative standard reduction be called an *administrative answer*. In the following, we rely on some known properties of the λ-calculus, which also apply to the administrative subset of the CPS λ-calculus.

Proposition 16. *Administrative reduction in λ_{cps}:*

1. *is* confluent*, so that if $M =_{ad} M'$, then there is some N such that $M \rightarrow^{\star}_{ad} N$ and $M' \rightarrow^{\star}_{ad} N$; and*

2. *has the* standardization *property, so that if $M \rightarrow^{\star}_{ad} N$ and N is an administrative answer, then there is an administrative answer M' such that $M \mapsto^{\star}_{ad} M' \rightarrow^{\star}_{ad} N$.*

Next we need to know how non-standard, or *internal*, administrative reductions interact with proper standard reductions. In short, they don't—administrative reductions commute with proper standard reductions (see Fig. 29).

$$\begin{tikzcd}
M \arrow[r, mapsto, dashed, "pr"] \arrow[d, "ad"'] & N' \arrow[d, dashed, "ad"] \\
M' \arrow[r, mapsto, "pr"] & N
\end{tikzcd}$$

(a) Proposition 17.1

$$\begin{tikzcd}
M \arrow[r, mapsto, dashed, "pr"] & N' \\
M' \arrow[u, "ad"] \arrow[r, mapsto, "pr"] & N \arrow[u, dashed, "ad"']
\end{tikzcd}$$

(b) Proposition 17.2

$$\begin{tikzcd}
M \arrow[r, mapsto, "pr"] \arrow[d, equal, "ad"'] & N \arrow[d, equal, dashed, "ad"] \\
M' \arrow[r, mapsto, dashed, "pr", "+"'] & N'
\end{tikzcd}$$

(c) Lemma 18

FIGURE 29. Diagrams of Proposition 17 and Lemma 18.

Proposition 17.

1. *If $M \to^\star_{ad} M' \mapsto_{pr} N$ and M is an administrative answer, then there is N' with $M \mapsto_{pr} N' \to^\star_{ad} N$.*

2. *If $M \leftarrow^\star_{ad} M' \mapsto_{pr} N$, then there is N' with $M \mapsto_{pr} N' \leftarrow^\star_{ad} N$.*

Proof.

1. If M is an administrative answer, then it is a proper β-redex; let $M \triangleq (\lambda(x^+).\, P)(V^+)$. Any administrative reductions in M must take place either in P or in V^+; in general, they could take P to some P' and V^+ to some V'^+. Hence $M' \equiv (\lambda(x^+).\, P')(V'^+)$. Since $M' \mapsto_{pr} N$, this means $N \equiv P'\{V'^+/x^+\}$, and we take $N' \triangleq P\{V^+/x^+\}$.

2. Similar. □

As a consequence, we have that $=_{ad}$, which can involve arbitrary administrative reductions in either direction, commutes with proper standard reduction.

Lemma 18. *If $M =_{ad} M' \mapsto_{pr} N$, then there is N' such that $M \mapsto^+_{pr1} N' =_{ad} N$.*

This is a crucial lemma; we will prove it later.

Now that we know what administrative reductions *don't* do, we should see what they *can* do: They serve to bring the standard redex in the source term to the top of the CPS term. This entails reifying the evaluation context as a continuation, so that we begin with $\mathcal{C}[\![E[M]]\!]\,\mathbf{ret}$ and build toward $\mathcal{C}[\![M]\!]K$, where K is a continuation that "represents" E somehow. We can formalize this intuition:

Proposition 19. *For each evaluation context E in λ_n or λ_v and each continuation K, there is a continuation K' such that for every term M we have $\mathcal{C}[\![E[M]]\!]K \mapsto^{\star}_{ad} \mathcal{C}[\![M]\!]K'$.*

Proof. By induction on the structure of E in each calculus. For call-by-name, we have two cases:

- If $E \equiv \square$, take $K' \triangleq K$.
- For $E \equiv E'N$, we have:

$$\begin{aligned}\mathcal{C}[\![E'[M]N]\!]K &\mapsto_{ad} \mathcal{C}[\![E'[M]]\!](\bar{\lambda}v.\, v(\mathcal{C}[\![N]\!], K)) \\ &\mapsto^{\star}_{ad} \mathcal{C}[\![M]\!]K' && \text{(by I.H.)}\end{aligned}$$

For call-by-value, we have three cases:

- If $E \equiv \square$, take $K' \triangleq K$.
- For $E \equiv E'N$, we have:

$$\begin{aligned}\mathcal{C}[\![E'[M]N]\!]K &\mapsto_{ad} \mathcal{C}[\![E'[M]]\!](\bar{\lambda}v.\, \mathcal{C}[\![N]\!](\bar{\lambda}w.\, v((\bar{\lambda}k'.\, k'w), K))) \\ &\mapsto^{\star}_{ad} \mathcal{C}[\![M]\!]K' && \text{(by I.H.)}\end{aligned}$$

– Finally, suppose $E \equiv V E'$. In our modified CBV calculus, V must be a λ-abstraction and not a variable, so we have:

$$\begin{aligned}
&\mathcal{C}[\![(\lambda x.\, N)(E'[M])]\!]K \\
&\quad \mapsto_{ad} (\bar{\lambda} k.\, k(\lambda(x, k').\, \mathcal{C}[\![N]\!]k'))(\bar{\lambda} v.\, \mathcal{C}[\![E'[M]]\!](\bar{\lambda} w.\, v((\bar{\lambda} k''.\, k''w), K))) \\
&\quad \mapsto^{\star}_{ad} \mathcal{C}[\![E'[M]]\!](\bar{\lambda} w.\, (\bar{\lambda} k.\, k(\lambda(x, k').\, \mathcal{C}[\![N]\!]k'))((\bar{\lambda} k''.\, k''w), K))
\end{aligned}$$

Note that if V could be a variable, then the CPS transformation would get stuck after the first step, and we would not be able to bring $E'[M]$ to the top of the transformed term. □

We now want to show that observations in the λ_n and λ_v calculi line up with observations of the CPS-transformed terms. In other words, we prove that $\mathcal{C}$ meets Criteria 13 to 15.

We define our forward simulation $\sim$ as follows:

Definition 20. *For a λ-term M (either CBN or CBV) and CPS term P, let $M \sim P$ when $\mathcal{C}[\![M]\!]\,\mathbf{ret} =_{ad} P$.*

Our task is to prove that $\sim$ satisfies the diagrams in (4.6), making it a forward simulation. To begin, we first prove that “answerness” and “stuckness” are preserved by administrative operations:

Proposition 21. *If $P =_{ad} P'$ and AP, then $P' \Downarrow$.*

Proof. By confluence, there must be a term Q such that $P \to^{\star}_{ad} Q$ and $P' \to^{\star}_{ad} Q$. Since P is an answer it must have the form $\mathbf{ret}\, V$, therefore the reductions in $\to^{\star}_{ad}$ must have been within V, so Q must have the form $\mathbf{ret}\, V'$. Finally, by standardization, since $P' \to^{\star}_{ad} \mathbf{ret}\, V'$, there must be R with $P' \mapsto^{\star}_{ad} R \to^{\star}_{ad} \mathbf{ret}\, V'$; since non-standard reductions cannot disturb the top redex, we must have $R \equiv \mathbf{ret}\, V''$, so $P' \Downarrow$. □

Proposition 22. *If $P =_{ad} P'$ and sP, then $P' \not\Downarrow$.*

Proof. Similar to Proposition 21, again invoking confluence and standardization. □

We are now ready to prove the third and forth commuting diagrams of Eq. (4.6), showing that $\sim$ relates answers to answers and stuck terms to stuck terms (up to some remaining steps in the CPS term):

Now we can prove the third and fourth commuting diagrams of (4.6):

Lemma 23. *If $M \sim P$ and AM, then $P \Downarrow$.*

Proof. Let $M \triangleq \lambda x.\, N$ for some N. Since $(\lambda x.\, N) \sim P$, we know that $P =_{ad} \mathcal{C}[\![\lambda x.\, N]\!]\, \mathbf{ret} =_{ad} \mathbf{ret}(\lambda(x, k').\, \mathcal{C}[\![]\!] N k')$, so the result is immediate by Proposition 21. □

Lemma 24. *If $M \sim P$ and sM, then $P \not\Downarrow$.*

Proof. A stuck λ-term, in either CBN or CBV, is one of the form $E[x]$ for some free x. By Proposition 19, $\mathcal{C}[\![E[x]]\!]\, \mathbf{ret} \mapsto^{\star}_{ad} \mathcal{C}[\![x]\!] K \mapsto_{ad} sxK$. So $\mathcal{C}[\![E[x]]\!]\, \mathbf{ret} \not\Downarrow$, and hence by Proposition 22, $P \not\Downarrow$. □

We also have the first commuting diagram, showing that $\sim$ relates a source term to its translation:

Lemma 25. $M \sim \mathcal{C}[\![M]\!]\, \mathbf{ret}$.

Proof. Unfolding definitions, we need that $\mathcal{C}[\![M]\!]\, \mathbf{ret} =_{ad} \mathcal{C}[\![]\!] M\, \mathbf{ret}$, which is immediate since $=_{ad}$ is reflexive. □

To show correctness of $\mathcal{C}$ it remains to satisfy the second commuting diagram, showing that our invariant $\sim$ is preserved under reduction. We prove this in three steps:

1. Show that, if $M \mapsto N$ by a β-reduction in the empty context, then we have $\mathcal{C}[\![M]\!]K \mapsto^{\star} =_{ad} \mathcal{C}[\![N]\!]K$ (Proposition 27).

2. Allow the reduction $M \mapsto N$ to occur in any evaluation context, not only at the top of the term (Proposition 28).

3. Let the CPS term be *any* $P =_{ad} \mathcal{C}[\![M]\!]\,\mathbf{ret}$ (Lemma 29).

We can get the first step using a simple proposition concerning substitution:

Proposition 26.

1. $\mathcal{C}[\![M]\!]\{\mathcal{C}[\![N]\!]/x\} \rightarrow^{\star}_{ad} \mathcal{C}[\![M\{N/x\}]\!]$

2. $\mathcal{C}[\![M]\!]\{(\bar{\lambda}k.\,\mathcal{C}[\![N]\!]k)/x\} \rightarrow^{\star}_{ad} \mathcal{C}[\![M\{N/x\}]\!]$

Proof.

1. By induction on the structure of M. The most interesting case is when $M \equiv x$, and thus we actually substitute N for x:

$$\mathcal{C}[\![x]\!]\{\mathcal{C}[\![N]\!]/x\} \triangleq \bar{\lambda}k.\,\mathcal{C}[\![N]\!]k$$

By inspection of $\mathcal{C}$, $\mathcal{C}[\![N]\!]$ must be some administrative λ-abstraction of the form $(\bar{\lambda}k'.\,P)$:

$$\begin{aligned} &\triangleq \bar{\lambda}k.\,(\bar{\lambda}k'.\,P)k \\ &\rightarrow_{ad} \bar{\lambda}k.\,(P\{k/k'\}) \\ &\equiv \bar{\lambda}k'.\,P \\ &\equiv \mathcal{C}[\![N]\!] \end{aligned}$$

2. Similar, only with one extra reduction at the beginning:

$$\begin{aligned}
\mathcal{C}[\![x]\!]\{\bar{\lambda}k.\,\mathcal{C}[\![N]\!]k/x\} &\triangleq (\bar{\lambda}k.\,xk)\{(\bar{\lambda}k.\,\mathcal{C}[\![N]\!]k)/x\} \\
&\triangleq (\bar{\lambda}k.\,(\bar{\lambda}k.\,\mathcal{C}[\![N]\!]k)k) \\
&\to_{ad} \bar{\lambda}k.\,\mathcal{C}[\![N]\!]k \\
&\to_{ad} \mathcal{C}[\![N]\!] \text{ (as before)} \qquad \square
\end{aligned}$$

And now:

Proposition 27. *If $M \mapsto N$ by a reduction in the empty context, then $\mathcal{C}[\![M]\!]K \mapsto^{+}_{pr^1} =_{ad} \mathcal{C}[\![N]\!]K$.*

Proof. Since the reduction takes place at the top level, we must have that M is a redex. From here, we must consider CBN and CBV separately:

– In CBN, M must have the form $(\lambda x.\,M')N'$ with $N \equiv M'\{N'/x\}$, and:

$$\begin{aligned}
\mathcal{C}[\![M]\!]K &\equiv \mathcal{C}[\![(\lambda x.\,M')N']\!]K \\
&\mapsto^{\star}_{ad} (\lambda(x,k').\,\mathcal{C}[\![M]\!]k')(\lambda k.\,\mathcal{C}[\![N]\!]k, K) \\
&\mapsto_{pr} (\mathcal{C}[\![M]\!]k)\{\lambda k.\,\mathcal{C}[\![N]\!]k/x\} \\
&\to^{\star}_{ad} \mathcal{C}[\![M'\{N'/x\}]\!]K \qquad \text{(by Proposition 26)} \\
&\equiv \mathcal{C}[\![N]\!]K
\end{aligned}$$

– In CBV, M must have the more specific form $(\lambda x.\, M')V$ with $N \equiv M'\{V/x\}$. Let $V \triangleq \lambda y.\, N'$.

$$\begin{aligned}
\mathcal{C}[\![M]\!]K &\equiv \mathcal{C}[\![(\lambda x.\, M')(\lambda y.\, N')]\!]K \\
&\mapsto^{\star}_{ad} (\lambda(x,k').\, \mathcal{C}[\![M']\!]k')((\bar{\lambda} h.\, h(\lambda(y,h').\, \mathcal{C}[\![N']\!]h')), K) \\
&\triangleq (\lambda(x,k').\, \mathcal{C}[\![M']\!]k')(\mathcal{C}[\![\lambda y.\, N']\!], K) \\
&\triangleq (\lambda(x,k').\, \mathcal{C}[\![M']\!]k')(\mathcal{C}[\![V]\!], K) \\
&\mapsto_{pr} \mathcal{C}[\![M']\!]\{\mathcal{C}[\![V]\!]/x\}K \\
&\to^{\star}_{ad} \mathcal{C}[\![M'\{V/x\}]\!]K && \text{(by Proposition 26)} \\
&\equiv \mathcal{C}[\![N]\!]K && \square
\end{aligned}$$

The second step is to show that a reduction in an evaluation context is performed faithfully by the CPS-transformed term.

Proposition 28. *If $M \mapsto N$, then $\mathcal{C}[\![M]\!]K \mapsto^{+}_{pr1} =_{ad} \mathcal{C}[\![N]\!]K$.*

Proof. By definition of $\mapsto$, we have that $M \equiv E[M']$ and $N \equiv E[N']$, where $M' \mapsto N'$ at top level.

$$\begin{aligned}
\mathcal{C}[\![M]\!]\,\mathbf{ret} &\equiv \mathcal{C}[\![E[M']]\!]K \\
&\mapsto^{\star}_{ad} \mathcal{C}[\![M']\!]K' \text{ (by Proposition 19)} \\
&\mapsto^{+}_{pr1} =_{ad} \mathcal{C}[\![N']\!]K' \text{ (by Proposition 27)} \\
&\leftarrow^{\star}_{ad} \mathcal{C}[\![E[N']]\!]\,\mathbf{ret} \ \text{ (property of } K' \text{ from Proposition 19)} \\
&\equiv \mathcal{C}[\![N]\!]K && \square
\end{aligned}$$

The third step of the proof is the most difficult: We must generalize the hypothesis of Proposition 28 so that it may be chained through multiple reduction steps. This hinges on the commutation lemma, which we now prove:

$$\begin{tikzcd}
M \arrow[r, equal, "ad"] & M' \arrow[r, mapsto, "pr"] & N
\end{tikzcd}$$

(a) Hypothesis

$$\begin{tikzcd}
 & M' \arrow[r, mapsto, "pr"] \arrow[d, dashed, "ad"] & N \\
M \arrow[ur, equal, "ad"] \arrow[r, dashed, "ad"] & M'' &
\end{tikzcd}$$

(b) Confluence

$$\begin{tikzcd}
 & M' \arrow[r, mapsto, "pr"] \arrow[d, "ad"] & N \\
M \arrow[ur, equal, "ad"] \arrow[r, "ad"] \arrow[dr, dashed, mapsto, "ad"'] & M'' & \\
 & M''' \arrow[u, dashed, "ad"'] &
\end{tikzcd}$$

(c) Standardization

$$\begin{tikzcd}
 & M' \arrow[r, mapsto, "pr"] \arrow[d, "ad"] & N \arrow[d, dashed, "ad"] \\
M \arrow[ur, equal, "ad"] \arrow[r, "ad"] \arrow[dr, mapsto, "ad"'] & M'' \arrow[r, dashed, mapsto, "pr"] & \cdot \\
 & M''' \arrow[u, "ad"'] \arrow[r, dashed, mapsto, "pr"] & N' \arrow[u, dashed, "ad"']
\end{tikzcd}$$

(d) Commutation

FIGURE 30. A summary of the proof of Lemma 18.

Proof (of Lemma 18). By Proposition 16.1, we know there must be some M'' such that $M \twoheadrightarrow^{\star}_{ad} M'' \twoheadleftarrow^{\star}_{ad} M'$. By Proposition 17, M'' can take a proper standard reduction, so Proposition 16.2 applies, giving M''' with $M \mapsto^{\star}_{ad} M''' \twoheadrightarrow^{\star}_{ad} M''$. From there, we use Proposition 17 to "fill in the arrows" (see Fig. 30). □

From there, we have that reduction preserves the invariant $\sim$:

Lemma 29. *If $M \mapsto N$ and $M \sim P$, then there is Q such that $P \mapsto^{+} Q$ and $N \sim Q$.*

Proof. Since $M \sim P$, we have $P =_{ad} \mathcal{C}[\![]\!] M\,\mathbf{ret}$. By Proposition 28, then, $\mathcal{C}[\![M]\!]\,\mathbf{ret} \mapsto^{+}=_{ad} \mathcal{C}[\![]\!] N\,\mathbf{ret}$, so $P =_{ad}\mapsto^{+}_{pr1}=_{ad} \mathcal{C}[\![]\!] N\,\mathbf{ret}$. Since $\mapsto^{+}_{pr1}$ is short for $\mapsto^{\star}_{ad}\mapsto_{pr}\mapsto^{\star}_{ad}$, we have $P =_{ad}\mapsto_{pr}=_{ad} \mathcal{C}[\![]\!] N\,\mathbf{ret}$. By Lemma 18, we have Q such that $P \mapsto^{+} Q =_{ad} \mathcal{C}[\![]\!] N\,\mathbf{ret}$, so $P \mapsto^{+} Q$ and $N \sim Q$. □

With all the pieces of (4.6) in hand, we can show that the uniform CPS transform preserves observations forward:

Theorem 30. *For any λ-term M and variable k:*

1. *If $M \Downarrow$ then $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$.*

2. *If $M \Uparrow$ then $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Uparrow$.*

3. *If $M \not\Downarrow$ then $\mathcal{C}[\![M]\!]\,\mathbf{ret} \not\Downarrow$.*

Proof. By Lemma 25, we can generalize $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ to any P with $M \sim P$. From there, 1 and 3 follow by induction on the reduction sequence and Lemmas 23, 24, and 29. 2 is immediate from Lemma 29, since P must take at least as many steps as M. □

Corollary 31. *For any λ-term M and variable k:*

1. *$M \Downarrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$.*

2. *$M \Uparrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Uparrow$.*

3. *$M \not\Downarrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \not\Downarrow$.*

Proof. The forward directions are Theorem 30. For the backward direction of 1, assume M does not reduce to an answer. Then it must either diverge or become stuck. By Theorem 30, in either case, $\mathcal{C}[\![M]\!]\,\mathbf{ret}$ must do the same, and thus (by determinism) it cannot reduce to an answer; by contraposition, $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$ implies $M \Downarrow$. The other clauses are similar. □

Correctness of the Naming Transform

Passing names instead of values has a subtle effect on the execution of a program: The CPS terms now express *sharing*. Since values aren't copied but are shared among subterms, relating reductions of unnamed terms to those of named terms requires care. For instance, consider this CPS term:

$$M \triangleq (\lambda x.\, f(x, x))(\lambda x.\, N)$$

It β-reduces and duplicates $(\lambda x.\,N)$:

$$M \mapsto f(\lambda x.\,N, \lambda x.\,N)$$

Now consider M under the naming transform:

$$\mathcal{N}[\![M]\!] \triangleq \nu y.\, y \coloneqq \lambda x.\mathcal{N}[\![N]\!] \;\mathbf{in}\; (\lambda x.\, f(x,x))y$$

It only duplicates the name y:

$$\mathcal{N}[\![M]\!] \mapsto \nu y.\, y \coloneqq \lambda x.\mathcal{N}[\![N]\!] \;\mathbf{in}\; f(y,y)$$

Notice, however, that if we reduce M and *then* translate, we get something different:

$$\mathcal{N}[\![f(\lambda z.\,N, \lambda z.\,N)]\!] \triangleq \nu x.\, x \coloneqq \lambda z.\mathcal{N}[\![N]\!] \;\mathbf{in}\; \nu y.\, y \coloneqq \lambda z.\mathcal{N}[\![N]\!] \;\mathbf{in}\; f(x,y)$$

Now there is no sharing of the value $\lambda z.\,N$.

In short, reduction does not *commute* with naming: Reducing the named term can produce shared references that do not appear when naming the reduced term. However, differences in sharing do not affect the outcome of the computation. Therefore we seek a way to reason *up to sharing*—that is, we want to consider a term with the same computational content, but more sharing, as “close enough.” A straightforward way to remove sharing from the picture is to consider terms under a *readback* function that “flattens” a term’s bound variables, returning it to the unnamed form:

$$\mathcal{N}^{-1}\langle\!\langle f(x^+)\rangle\!\rangle \triangleq f(x^+)$$
$$\mathcal{N}^{-1}\langle\!\langle \nu x.\, x \coloneqq \lambda(x^+).\, N \textbf{ in } M\rangle\!\rangle \triangleq \mathcal{N}^{-1}\langle\!\langle M\rangle\!\rangle\{\lambda(x^+).\mathcal{N}^{-1}\langle\!\langle N\rangle\!\rangle/x\}$$

We can now define our invariant as follows:

Definition 32. *For an unnamed CPS term M and a named CPS term P, let $M \sim P$ when $M \equiv \mathcal{N}^{-1}\langle\!\langle P\rangle\!\rangle$.*

To show correctness of the naming step we will show that $\sim$ is a *backward* simulation: If $M \sim P$ and $P \mapsto P'$ then there is M' with $M \mapsto^\star M'$ and $M' \sim P'$. Note that, because *deref* reductions disappear under $\mathcal{N}^{-1}$, the correspondence between reductions is not one-to-at-least-one, as it was in (4.6). We relied on this property in proving Theorem 30.2, so we will have to adjust our reasoning this time.

First, we prove that reductions are preserved:

Proposition 33. *Given a $\lambda_{cps,vn}$ term P:*

1. *If $P \mapsto_\beta P'$, then $\mathcal{N}^{-1}\langle\!\langle P\rangle\!\rangle \mapsto \mathcal{N}^{-1}\langle\!\langle P'\rangle\!\rangle$.*

2. *If $P \mapsto_{deref} P'$, then $\mathcal{N}^{-1}\langle\!\langle P\rangle\!\rangle \equiv \mathcal{N}^{-1}\langle\!\langle P'\rangle\!\rangle$.*

Proof. Given any context E in the named CPS language, let σ_E be the substitution built up by the unnaming function as it traverses E. (In other words, σ_E is the substitution such that $\mathcal{N}^{-1}\langle\!\langle E[P]\rangle\!\rangle \equiv \mathcal{N}^{-1}\langle\!\langle P\rangle\!\rangle\sigma_E$.)

1. If $P \mapsto P'$ by the β rule, we have that:

$$\begin{aligned}
P &\equiv E[(\lambda(x^+).\, Q)(y^+)] \\
P' &\equiv E[Q\{y^+/x^+\}] \\
\mathcal{N}^{-1}\langle\!\langle P\rangle\!\rangle &\triangleq (\lambda(x^+).\, \mathcal{N}^{-1}\langle\!\langle Q\rangle\!\rangle)(y^+)\sigma_E \\
&\mapsto \mathcal{N}^{-1}\langle\!\langle Q\rangle\!\rangle\{y^+/x^+\}\sigma_E \\
&\equiv \mathcal{N}^{-1}\langle\!\langle E[Q\{y^+/x^+\}]\rangle\!\rangle \\
&\equiv \mathcal{N}^{-1}\langle\!\langle P'\rangle\!\rangle
\end{aligned}$$

2. If $P \mapsto P'$ by the *deref* rule, we have that:

$$\begin{aligned}
P &\equiv E[\nu f.\, f \coloneqq \lambda(x^+).\, Q \;\mathbf{in}\; E'[f(y^+)]] \\
P' &\equiv E[\nu f.\, f \coloneqq \lambda(x^+).\, Q \;\mathbf{in}\; E'[(\lambda(x^+).\, Q(y^+)]] \\
\mathcal{N}^{-1}\langle\!\langle P\rangle\!\rangle &\triangleq f(y^+)\sigma_{E'}\{\lambda(x^+).\, Q/f\}\sigma_E \\
&\equiv (\lambda(x^+).\, Q)(y^+)\sigma_{E'}\{\lambda(x^+).\, Q/f\}\sigma_E \\
&\equiv \mathcal{N}^{-1}\langle\!\langle P'\rangle\!\rangle
\end{aligned}$$ □

Now we can show that $\mathcal{N}^{-1}$ preserves observable behavior:

Theorem 34.

1. *If $\mathcal{N}[\![M]\!] \Downarrow$ then $M \Downarrow$.*

2. *If $\mathcal{N}[\![M]\!] \Uparrow$ then $M \Uparrow$.*

3. *If $\mathcal{N}[\![M]\!] \not\Downarrow$ then $M \not\Downarrow$.*

Proof. The initial condition $M \sim \mathcal{N}[\![M]\!]$ follows from the fact that $\mathcal{N}$ and $\mathcal{N}^{-1}$ form a retraction pair: for any unnamed CPS term M, $\mathcal{N}^{-1}\langle\!\langle \mathcal{N}[\![M]\!]\rangle\!\rangle \equiv M$. The invariant holds by Proposition 33. Since answers and stuck terms are virtually the same between unnamed and named terms, the final conditions are trivial. It only remains to show that divergence is preserved. This is easy, however, as each *deref*-reduction always produces a β-redex, so M and $\mathcal{N}[\![M]\!]$ must take the same number of β-reductions. □

As we did for the uniform CPS transform (only in reverse), we get the forward directions from the backward ones, completing the correctness proof.

Corollary 35.

1. *$M \Downarrow$ iff $\mathcal{N}[\![M]\!] \Downarrow$.*
2. *$M \Uparrow$ iff $\mathcal{N}[\![M]\!] \Uparrow$.*
3. *$M \not\Downarrow$ iff $\mathcal{N}[\![M]\!] \not\Downarrow$.*

Proof. The backward directions are Theorem 34. The forward directions use case analysis and determinism in the same way as Corollary 31. □

4.4 CPS and Processes

The π-calculus describes computation as the exchange of simple messages by independent agents, called *processes*. Each term in the π-calculus describes a process, and processes are built by *composing* them together in parallel, *prefixing* them with I/O actions, and *replicating* them. Communication takes place over *channels*, each of which has a *name*; processes interact when one is writing to a channel and, in parallel, another is reading from it. The values sent over the channels are themselves channel names, so processes can discover each other dynamically. Names act much like variables in the λ-calculus, with α-equivalent terms identified in the same way. They can be allocated by the $\boldsymbol{\nu}$ construct, which guarantees that the name it binds will be distinct from any other allocated or free name.

$$\text{Processes:}\quad P, Q ::= \overline{x}\langle y^+\rangle \;\big|\; x(y^+).\, P \;\big|\; (P \mid Q) \;\big|\; !P \;\big|\; \boldsymbol{\nu} z\, P$$

$$\frac{}{\overline{x}\langle y^+\rangle \mid x(z^+).\, P \to P\{y^+/z^+\}}$$

$$\frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q}$$

$$\frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad \frac{P \to P'}{\boldsymbol{\nu} z\, P \to \boldsymbol{\nu} z\, P'}$$

$$\begin{array}{cc}
P \mid Q \equiv Q \mid P & !P \equiv {!P} \mid P \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) & (\boldsymbol{\nu} z\, P) \mid Q \equiv \boldsymbol{\nu} z\, (P \mid Q) \\
\boldsymbol{\nu} x\, \boldsymbol{\nu} y\, P \equiv \boldsymbol{\nu} y\, \boldsymbol{\nu} x\, P & \text{if } z \text{ not free in } Q
\end{array}$$

FIGURE 31. A fragment of the π-calculus.

The syntax and semantics for the fragment of the π-calculus we are considering are given in Fig. 31. This fragment is called the *asynchronous π-calculus* because there are no processes of the form $\overline{x}\langle y\rangle.\, P$. In other words, no process is ever blocked waiting for a write operation to complete. This property reflects the behavior of CPS terms: They never wait for a subterm to compute, instead providing a continuation that performs the remaining work.

Processes in the π-calculus are meant to be considered up to a relation called *structural congruence*, which we write as $\equiv$.[5] The rules (other than those making $\equiv$ an equivalence relation and a congruence) are given in Fig. 31. Reductions are closed up to structural congruence (as well as parallel composition and name allocation). Besides eliminating unimportant differences such as the order of parallel composition, structural congruence accounts for the spawning of replicated processes and the scoping of allocated names.

[5]In the π-calculus literature, structural congruence is usually written as $\equiv$.

π from CPS

Despite the radically different approaches to expressing computation, CPS transforms and π-encodings are interrelated. In particular, given a CPS transform, one can systematically *derive* a π-encoding from it (Sangiorgi, 1999; Sangiorgi & Walker, 2003). The major difficulty in deriving a π-encoding from a CPS transform arises from an important difference: Functions in the λ-calculus can take functions as arguments, but processes in the π-calculus do not send processes over channels, only names. In other words, λ is higher-order, but π is first-order.

But we have already addressed this mismatch: The *value-named* CPS language $\lambda_{cps,vn}$ is "first-order" in much the same way. In fact, nearly every construct in the named CPS calculus $\lambda_{cps,vn}$ corresponds directly to a construct in the π-calculus:

– An application $x(y^+)$ becomes a process $\overline{x}\langle y^+\rangle$, which performs a *write* on channel x, then halts. The tuple (y^+) is transmitted over x.

– Each binding $\nu x.\, x \coloneqq \lambda(y^+).\, N \ \mathbf{in}\ M$ becomes a process of the form $\boldsymbol{\nu} x\ (P \mid\ !x(y^+).\, Q)$. This process allocates a fresh channel name x, then runs a process P in parallel with the process $!x(y^+).\, Q$. The latter acts as a "server": It listens on the channel x for a request, then runs the process Q with the request's values as arguments. The ! makes the server process replicated, so that it handles any number of requests over time.

The only terms without couterparts are applications with λ-abstractions in head position—that is, β-redexes. But we can handle these by reducing them during translation.

Thus we can faithfully translate $\lambda_{cps,vn}$ to the π-calculus:

$$\begin{aligned}
\mathcal{P}[\![V(y^+)]\!] &\triangleq \mathcal{P}[\![V]\!]_{y^+} \\
\mathcal{P}[\![\nu x.\, x \coloneqq \lambda(y^+).\, N \,\mathbf{in}\, M]\!] &\triangleq \boldsymbol{\nu} x\, (\mathcal{P}[\![M]\!] \mid !x(y^+).\, \mathcal{P}[\![N]\!]) \\
\mathcal{P}[\![f]\!]_{y^+} &\triangleq \overline{f}\langle y^+\rangle \\
\mathcal{P}[\![\mathbf{ret}]\!]_{y^+} &\triangleq \overline{\mathbf{ret}}\langle y^+\rangle \\
\mathcal{P}[\![\lambda(x^+).\, M]\!]_{y^+} &\triangleq \mathcal{P}[\![M]\!]\{y^+/x^+\}
\end{aligned}$$

The subscripted form of $\mathcal{P}$ translates a term, given the arguments it is being applied to, performing β-reduction as needed. To translate **ret**, we simply assume some fresh π-calculus channel name **ret**.

Finally, we obtain the π-calculus encoding (Fig. 32) by running the uniform CPS transform $\mathcal{C}$ through the naming transform $\mathcal{N}$ and then through the π-calculus translation $\mathcal{P}$. The final product coincides with the established uniform π-encoding (Sangiorgi & Walker, 2003).[6]

Correctness

For a π-calculus term P, if P is capable of performing a write on the free channel name k (possibly after some reductions), we write $P \Downarrow_{\overline{k}}$. A named CPS term signals termination by invoking initial continuation **ret**, which is translated to a write on **ret**. Hence we expect the π-encoded term to write on **ret** if and only if the named CPS term would invoke **ret**:

Lemma 36. *For any $\lambda_{cps,vn}$ term M, $M \Downarrow$ iff $\mathcal{P}[\![M]\!] \Downarrow_{\overline{\mathbf{ret}}}$.*

[6]In fact, the final transform in Fig. 32 differs slightly from the uniform π-encoding in the literature, in that all input processes are replicated, even those used at most once (e.g.continuation processes). However, this is harmless, as garbage collection is sound in the π-calculus (up to bisimulation). The call-by-need π-encoding will keep some processes unreplicated, as this is necessary for correctness.

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, k(\lambda(x,k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \begin{cases} \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, v(\lambda k'.\, \mathcal{C}[\![N]\!]k', k)) & \text{CBN} \\ \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \mathcal{C}[\![N]\!](\lambda w.\, v(\lambda k'.\, k'w, k))) & \text{CBV} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{vn}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}_{vn}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, \nu f.\, f := \lambda(x,k').\, \mathcal{C}_{vn}[\![M]\!]k' \;\mathbf{in}\; kf \\
\mathcal{C}_{vn}[\![MN]\!] &\triangleq \begin{cases} \begin{array}{l} \lambda k.\, \nu k'.\, k' := (\lambda v.\, \nu x.\, x := \lambda k'.\, \mathcal{C}_{vn}[\![N]\!]k' \;\mathbf{in} \\ \quad v(x,k)) \;\mathbf{in}\; \mathcal{C}_{vn}[\![M]\!]k' \end{array} & \text{CBN} \\ \begin{array}{l} \lambda k.\, \nu k'.\, k' := \left(\begin{array}{l} \lambda v.\, \nu k''.\, k'' := (\lambda w. \\ \quad \nu x.\, x := \lambda k'.\, k'w \;\mathbf{in} \\ \quad v(x,k)) \;\mathbf{in}\; \mathcal{C}_{vn}[\![N]\!]k'' \end{array} \right) \mathbf{in} \\ \quad \mathcal{C}_{vn}[\![M]\!]k' \end{array} & \text{CBV} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![x]\!]_k &\triangleq \overline{x}\langle k \rangle \\
\mathcal{E}[\![\lambda x.\, M]\!]_k &\triangleq \boldsymbol{\nu} f\; (\overline{k}\langle f \rangle \mid !f(x,k).\, \mathcal{E}[\![M]\!]_k) \\
\mathcal{E}[\![MN]\!]_k &\triangleq \begin{cases} \begin{array}{l} \boldsymbol{\nu} k'\; (\mathcal{E}[\![M]\!]_{k'} \mid \\ \quad !k'(v).\, \boldsymbol{\nu} x\; (\overline{v}\langle x,k \rangle \mid !x(k'').\, \mathcal{E}[\![N]\!]_{k''})) \end{array} & \text{CBN} \\ \begin{array}{l} \boldsymbol{\nu} k'\; (\mathcal{E}[\![M]\!]_{k'} \mid k'(v).\, \boldsymbol{\nu} k''\; (\mathcal{E}[\![N]\!]_{k''} \mid \\ \quad !k''(w).\, \boldsymbol{\nu} x\; (\overline{v}\langle x,k \rangle \mid !x(k').\, \overline{k'}\langle w \rangle)))) \end{array} & \text{CBV} \end{cases}
\end{aligned}
$$

FIGURE 32. The CPS transform, named CPS transform, and π-encoding for CBN and CBV.

Proof. We need only that our syntactic embedding of $\lambda_{cps,vn}$ into π is also a semantic embedding—in other words, that reductions in the CPS term correspond to reductions in the π-term and vice versa. The only catch is that β-redexes from the CPS term disappear under $\mathcal{P}$; however, $\mathcal{P}$ is still a bisimulation.

Note that any CPS term $E[M]$ will translate to processes representing the bindings in E in parallel with the process representing M. Thus we can consider translating E and M separately. Then:

$$\begin{aligned}&\nu f.\, f \coloneqq \lambda(x^+).\, M \,\mathbf{in}\, E[f(y^+)]\\ &\quad \mapsto^\star \nu f.\, f \coloneqq \lambda(x^+).\, M \,\mathbf{in}\, E[M\{y^+/x^+\}]\end{aligned}$$

corresponds to

$$\begin{aligned}&\boldsymbol{\nu} f\ (\overline{f}\langle y^+\rangle \mid \mathcal{P}[\![E]\!] \mid {!f(x^+)}.\, \mathcal{P}[\![M]\!])\\ &\quad \mapsto \boldsymbol{\nu} f\ (M\{y^+/x^+\} \mid \mathcal{P}[\![E]\!] \mid {!f(x^+)}.\, \mathcal{P}[\![M]\!]).\end{aligned}$$

It is straightforward to construct a grammar of possible π-terms produced by $\mathcal{P}$ to show that the correspondence works in both directions. $\square$

Finally, we have the complete proof of the correctness of the uniform π-encoding, simply by composing:

Theorem 37. *For any λ_n or λ_v term M, $M \Downarrow$ if and only if $\mathcal{E}[\![M]\!]_{\mathbf{ret}} \Downarrow_{\overline{\mathbf{ret}}}$.*

Proof. Note that $\mathcal{E}[\![M]\!]_{\mathbf{ret}} \equiv \mathcal{P}[\![\mathcal{N}[\![\mathcal{C}[\![M]\!]\,\mathbf{ret}]\!]]\!]$; the result follows by Corollaries 31 and 35 and Lemma 36. $\square$

Uniform Abstract Machine

In addition to the π-encoding, we can derive other artifacts from the uniform CPS transform. In particular, through the functional correspondence (Ager, Danvy, & Midtgaard, 2004), we obtain a uniform abstract machine for CBN and CBV. See Fig. 33.

The CBN fragment of the machine is essentially the same as the well-known Krivine machine for CBN evaluation (Krivine, 2007). On closed terms, the CBV fragment strongly resembles the CEK machine (Felleisen & Friedman, 1986) (without control operators). Unlike most environment-based abstract machines, ours does not exclude open terms, and thus its behavior can meaningfully be more finely specified: Variable lookup happens when a variable is evaluated, and only λ-abstractions are treated as values. We could instead delay the variable lookup until a λ-abstraction is required; this machine would implement the full CBV β-rule. Much as with the uniform CPS transform, we can observe the difference using a term such as $(\lambda x.\, \lambda y.\, y)z$, which is stuck according to the uniform abstract machine.

4.5 Call-by-Need and Constructive Update

As we have seen, CBV *usually* takes fewer evaluation steps to reach an answer than CBN. However, CBV evaluation wastes work whenever a function does not use its argument. The call-by-need λ-calculus (Ariola et al., 1995; Ariola & Felleisen, 1997) is efficient in both cases: Unneeded arguments are never computed, yet each argument is evaluated at most once. Hence call-by-need models efficient implementations of lazy evaluation, which *memoize*, or cache, each computed value.

The syntax and semantics are given in Fig. 34. Rather than perform a substitution, the β_{need} rule suspends the argument in a **let** binding. The grammar for evaluation contexts expresses lazy evaluation order: Given a term

$$\begin{aligned}
\text{Terms:} \quad M, N &::= x \mid \lambda x.\, M \mid MN \\
\text{Continuations:} \quad k &::= \mathit{KRet} \mid \mathit{KAppN} M, k, \rho \\
&\mid \mathit{KAppV1} M, k, \rho \\
&\mid \mathit{KAppV2} \lambda x.\, M, k, \rho \\
\text{Environments:} \quad \rho &::= \epsilon \mid \rho[x = \mathit{KClos} M, \rho] \\
\text{States:} \quad S &::= \langle M, k, \rho \rangle_M \mid \langle k, \lambda x.\, M, \rho \rangle_K \\
&\mid \langle \lambda x.\, M, \rho \rangle_H
\end{aligned}$$

$$M \mapsto \langle M, \mathit{KRet}, \epsilon \rangle_M$$

$$\begin{aligned}
\langle x, k, \rho \rangle_M &\mapsto \langle M, k, \rho' \rangle_M \\
\text{where } \rho(x) &\equiv \mathit{KClos} M, \rho' \\
\langle \lambda x.\, M, k, \rho \rangle_M &\mapsto \langle k, \lambda x.\, M, \rho \rangle_K \\
\langle MN, k, \rho \rangle_M &\mapsto \langle M, k', \rho \rangle_M \\
\text{where } k' &\triangleq \begin{cases} \mathit{KAppN} N, k, \rho & \text{CBN} \\ \mathit{KAppV1} N, k, \rho & \text{CBV} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\langle \mathit{KRet}, \lambda x.\, M, \rho \rangle_K &\mapsto \langle \lambda x.\, M, \rho \rangle_H \\
\langle \mathit{KAppN} N, k, \rho', \lambda x.\, M, \rho \rangle_K &\mapsto \langle M, k, \rho'' \rangle_M \\
\text{where } \rho'' &\triangleq \rho[x = \mathit{KClos} N, \rho'] \\
\langle \mathit{KAppV1} N, k, \rho', \lambda x.\, M, \rho \rangle_K &\mapsto \langle N, k', \rho' \rangle_M \\
\text{where } k' &\triangleq \mathit{KAppV2} \lambda x.\, M, k, \rho \\
\langle \mathit{KAppV2} \lambda y.\, N, k, \rho', \lambda x.\, M, \rho \rangle_K &\mapsto \langle N, k, \rho'' \rangle_M \\
\text{where } \rho'' &\triangleq \rho'[y = \mathit{KClos} \lambda x.\, M, \rho]
\end{aligned}$$

FIGURE 33. Uniform abstract machine for CBN and CBV.

$$\begin{array}{rrl}
\text{Expressions:} & M, N ::= & x \mid \lambda x.\, M \mid M N \mid \mathbf{let}\, x = M \,\mathbf{in}\, N \\
\text{Values:} & V ::= & \lambda x.\, M \\
\text{Answers:} & A ::= & V \mid \mathbf{let}\, x = M \,\mathbf{in}\, A \\
\text{Evaluation Contexts:} & E, F ::= & [] \mid E M \mid \mathbf{let}\, x = E \,\mathbf{in}\, F[x] \\
& & \mid \mathbf{let}\, x = M \,\mathbf{in}\, E
\end{array}$$

$$\begin{array}{rll}
(\lambda x.\, M) N & \to \mathbf{let}\, x = N \,\mathbf{in}\, M & \beta_{need} \\
(\mathbf{let}\, y = L \,\mathbf{in}\, A) N & \to \mathbf{let}\, y = L \,\mathbf{in}\, A N & \mathit{lift} \\
\mathbf{let}\, x = V \,\mathbf{in}\, E[x] & \to \mathbf{let}\, x = V \,\mathbf{in}\, E[V] & \mathit{deref} \\
\mathbf{let}\, x = (\mathbf{let}\, y = L \,\mathbf{in}\, A) \,\mathbf{in}\, E[x] & \to \mathbf{let}\, y = L \,\mathbf{in}\, \mathbf{let}\, x = A \,\mathbf{in}\, E[x] & \mathit{assoc}
\end{array}$$

FIGURE 34. The call-by-need λ-calculus, λ_{need}, of Ariola et al. (Ariola, Felleisen, Maraist, Odersky, & Wadler, 1995).

$\mathbf{let}\, x = M \,\mathbf{in}\, N$, we evaluate N until it becomes either a value or a term of the form $F[x]$ for some evaluation context F. Such a term *needs* the value of x to continue, so now we evaluate the term M that x is bound to. But since this computation is done in place, it only needs to be done once: After M becomes a value, this value will simply be substituted directly (by the *deref* rule) if x is needed again. If N instead becomes a value, then we say the whole term is an *answer*—a value surrounded by a number of **let** bindings.

An answer in call-by-need is "almost a value." Evaluation stops when a term becomes an answer, but it's not a value for the purposes of the β or *deref* rule. When a subterm evaluates to an answer, either the *lift* or the *assoc* rule moves each binding into the outer environment.

To illustrate, we turn to our previous example, $(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z))$. It reduces as follows:

$$
\begin{aligned}
&(\lambda x.\, xx)((\lambda y.\, y)(\lambda z.\, z)) \\
&\quad \mapsto_{\beta_{need}} \mathbf{let}\, x = (\lambda y.\, y)(\lambda z.\, z)\,\mathbf{in}\, xx \\
&\quad \mapsto_{\beta_{need}} \mathbf{let}\, x = (\mathbf{let!}y = \lambda z.\, z\,\mathbf{in}\, y)\,\mathbf{in}\, xx \\
&\quad \mapsto_{deref} \mathbf{let}\, x = (\mathbf{let}\, y = \lambda z.\, z\,\mathbf{in}\, \lambda z.\, z)\,\mathbf{in}\, xx \\
&\quad \mapsto_{assoc} \mathbf{let}\, y = \lambda z.\, z\,\mathbf{in}\,\mathbf{let!}x = \lambda z.\, z\,\mathbf{in}\, xx \\
&\quad \mapsto_{deref} \mathbf{let}\, y = \lambda z.\, z\,\mathbf{in}\,\mathbf{let}\, x = \lambda z.\, z\,\mathbf{in}\, (\lambda z.\, z)x \\
&\quad \mapsto_{\beta_{need}} \mathbf{let}\, y = \lambda z.\, z\,\mathbf{in}\,\mathbf{let!}x = \lambda z.\, z\,\mathbf{in}\,\mathbf{let}\, z = x\,\mathbf{in}\, z \\
&\quad \mapsto_{deref} \mathbf{let}\, y = \lambda z.\, z\,\mathbf{in}\,\mathbf{let}\, x = \lambda z.\, z\,\mathbf{in}\,\mathbf{let!}z = \lambda z.\, z\,\mathbf{in}\, z \\
&\quad \mapsto_{deref} \mathbf{let}\, y = \lambda z.\, z\,\mathbf{in}\,\mathbf{let}\, x = \lambda z.\, z\,\mathbf{in}\,\mathbf{let}\, z = \lambda z.\, z\,\mathbf{in}\, \lambda z.\, z
\end{aligned}
$$

Evaluation begins in a call-by-name manner, in the sense that the outer β-redex is reduced immediately. The argument is suspended in a **let** binding. Next, since x is in head position in the body, we need to evaluate it, and so we reduce the inner β-redex. However, since the reduction is done in place, this step will only be done once. In all, three β-reductions are performed, as few as is done by call-by-value.

Unlike call-by-value, when a function ignores its argument, call-by-need does not waste work. In extreme cases, call-by-value *never finishes* when call-by-name or call-by-need would. For example, consider the term

$$(\lambda x.\, \lambda y.\, y)\Omega,$$

where Ω is a term $(\lambda x.\, xx)(\lambda x.\, xx)$ that diverges. Call-by-name substitutes the Ω immediately (the substitution is trivial since x does not occur in the body). Call-by-need similarly suspends the Ω without attempting to evaluate it. Call-by-value insists on computing the argument first, and thus is caught in an infinite loop.

Continuation-Passing Style

There does exist a call-by-need CPS transform due to Okasaki et al. (Okasaki et al., 1994) It requires mutable storage, which our CPS languages do not support. However, suppose we borrow the assignment syntax from the named CPS language $\lambda_{cps,vn}$. Then we can build on the uniform CPS transform (Fig. 24) and use a call-by-need application rule:

$$\begin{aligned}\mathcal{C}[\![MN]\!] \triangleq \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \nu x.\, x &:= \\ &\big(\lambda k'.\, \mathcal{C}[\![N]\!](\lambda w.\, x := \lambda k''.\, k''w \;\mathbf{in}\; k'w)\big) \;\mathbf{in}\; v(x,k))\end{aligned}$$

This is roughly the same as in the Okasaki CPS transform. Unfortunately, this is not valid $\lambda_{cps,vn}$ syntax, as the assignment operator $:=$ is only allowed immediately inside a ν binding for the variable assigned to. As a result, $\lambda_{cps,vn}$ only allows an assignment to a variable that presently has no value. The inner continuation, $\lambda w.\, x := \lambda k''.\, k''w \;\mathbf{in}\; k'w$, violates this restriction by attempting to "overwrite" x. We will call this a *double assignment*.

Of course, this is precisely what we wish to happen: The term bound to x *should* change, in order to cache the computed value. But we don't need the full power of mutable storage; a much weaker effect will suffice.

To see this, suppose for a moment we allow $:=$ anywhere, with the semantics of destructive update (that is, each assignment overwrites any previous one). Inspecting the rule, we see that each variable is now assigned to (at most) twice: Once when it is initialized with a thunk, and again when the thunk's result is memoized. However, after the *second* assignment, the stored value never changes *again*. Furthermore, note that the initial thunk cannot refer to x, even indirectly, as x is not in the scope of the computation (our **let** is not recursive). Therefore the initial thunk is only used once; since that very thunk performs the second assignment, the first lookup must precede the second assignment, with no other accesses in between.

In the language of data-flow analysis, after the first lookup, x cannot be *live*. Hence its value does not matter. In other words, it may as well *have no value*. If we clear x after the first lookup, then the second assignment is just like the first: It is giving a value to a variable that currently has none. There is no double assignment.

This analysis suggests a special assignment operation that always clears the variable the next time it is used. The assigned value will therefore only be used once, and thus the assignment is *ephemeral*, as opposed to *permanent*. After a permanent assignment, the variable will never be cleared, so permanent assignments are final.

The Transform

Writing $x \coloneqq M \;\mathbf{in}\; N$ for a permanent assignment and $x \coloneqq_1 M \;\mathbf{in}\; N$ for an ephemeral assignment, we can modify the call-by-need CPS transform so that it does not require destructive update:

$$\begin{aligned}\mathcal{C}[\![MN]\!] \triangleq \lambda k.\,\mathcal{C}[\![M]\!](\lambda v.\,\nu x.\,x \coloneqq_1 \\ \big(\lambda k'.\,\mathcal{C}[\![N]\!](\lambda w.\,x \coloneqq \lambda k''.\,k''w \;\mathbf{in}\; k'w)\big) \;\mathbf{in}\; v(x,k))\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, k(\lambda(x,k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \nu x.\, x :=_1 \\
&\qquad \big(\lambda k'.\, \mathcal{C}[\![N]\!](\lambda w.\, x := \lambda k''.\, k''w \,\mathbf{in}\, k'w)\big) \\
&\quad\ \mathbf{in}\, v(x,k)) \\
\mathcal{C}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!] &\triangleq \lambda k.\, \nu x.\, x :=_1 \\
&\qquad \big(\lambda k'.\, \mathcal{C}[\![L]\!](\lambda w.\, x := \lambda k''.\, k''w \,\mathbf{in}\, k'w)\big) \\
&\quad\ \mathbf{in}\, \mathcal{C}[\![M]\!]k
\end{aligned}$$

FIGURE 35. A call-by-need CPS transform using constructive update.

Since the initial thunk is now assigned ephemerally, there is never a double assignment. In fact, we can prove so: By the above data-flow analysis, x is unassigned before the second assignment. Each assignment is performed by a term that is used at most once, and thus no further assignments will be attempted.

Note what has happened here: x takes on different values over time, due to multiple assignments. Therefore it is fair to say it was *updated.* However, no previous value was destroyed by any update, and in fact a previous value *cannot* be destroyed. In this language, updates only construct, never destroy; hence we call the phenomenon *constructive update.*

This CPS transform, summarized in Fig. 35, is the one we will relate to the π-calculus. The syntax and semantics for permanent and ephemeral assignment are given in Fig. 36. The deref_1 rule is similar to deref, only it removes the ephemeral assignment.

We have shown that terms produced by the call-by-need CPS transform never attempt a *double assignment*—that is, they never reduce to a term such as $x := V \,\mathbf{in}\, x := W \,\mathbf{in}\, M$. Let us call such terms *safe*:

$$\begin{array}{rrl}
\text{Terms:} & M, N ::= & V(V^+) \mid \nu x.\, M \\
& & \mid x := \lambda(x^+).\, M \mathbf{\ in\ } N \\
& & \mid x :=_1 \lambda(x^+).\, M \mathbf{\ in\ } N \\
\text{Values:} & V ::= & x \mid \mathbf{ret} \mid \lambda(x^+).\, M \\
\text{Binding Contexts:} & B ::= & \Box \mid \nu x.\, B \\
& & \mid x := \lambda(x^+).\, M \mathbf{\ in\ } B \\
& & \mid x :=_1 \lambda(x^+).\, M \mathbf{\ in\ } B \\
\text{Evaluation Contexts:} & E ::= & B
\end{array}$$

$$\begin{array}{rcll}
(\lambda(x^+).\, M)(V^+) & \rightarrow & M\{V^+/x^+\} & \beta \\
\begin{array}{l} f := \lambda(x^+).\, M \mathbf{\ in} \\ \quad E[f(V^+)] \end{array} & \rightarrow & \begin{array}{l} f := \lambda(x^+).\, M \mathbf{\ in} \\ \quad E[(\lambda(x^+).\, M)(V^+)] \end{array} & \textit{deref} \\
\begin{array}{l} f :=_1 \lambda(x^+).\, M \mathbf{\ in} \\ \quad E[f(V^+)] \end{array} & \rightarrow & E[(\lambda(x^+).\, M)(V^+)] & \textit{deref}_1
\end{array}$$

FIGURE 36. The CPS λ-calculus with constructive update, $\lambda_{cps}^{:=_1}$.

Definition 38. *A $\lambda_{cps}^{:=_1}$ term M is* safe *when it does not reduce to a term with a subterm of the form*

$$x :=_* V \mathbf{\ in\ } E[x :=_* W \mathbf{\ in\ } N],$$

where $:=_*$ stands for either $:=$ or $:=_1$ in each appearance.

Proposition 39. *For any M and K, if K is a variable,* **ret***, or $\lambda v.\, P$ where P is safe, then $\mathcal{C}[\![M]\!]K$ is safe.*

Proof. See the above data-flow analysis. $\Box$

Naming and π-Encoding

Now that we have the call-by-need CPS transform, we need only adapt the development in Sections and 4.4 to derive the π-calculus encoding.

In Section 4.1, we were somewhat sloppy in deriving a π-encoding from the CPS—all input processes in the encoding were replicated, even those that are provably used at most once. Now that we have ephemeral assignment, we can be more precise. Some λ-abstractions are *affine*, i.e. never duplicated; these are the ones representing continuations or suspended computations. We can mark the λs in the CPS transform to indicate which values are affine:

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, k(\lambda(x,k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \lambda k.\, \mathcal{C}[\![M]\!](\lambda_1 v.\, \nu x.\, x \coloneqq_1 \\
&\qquad\qquad \big(\lambda k'.\, \mathcal{C}[\![N]\!](\lambda_1 w.\, x \coloneqq \lambda k''.\, k''w \,\mathbf{in}\, k'w)\big) \\
&\qquad \mathbf{in}\, v(x,k)) \\
\mathcal{C}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!] &\triangleq \lambda k.\, \nu x.\, x \coloneqq_1 \\
&\qquad\quad \big(\lambda k'.\, \mathcal{C}[\![L]\!](\lambda_1 w.\, x \coloneqq \lambda k''.\, k''w \,\mathbf{in}\, k'w)\big) \\
&\qquad \mathbf{in}\, \mathcal{C}[\![M]\!]k
\end{aligned}
$$

Now we augment the naming transform to treat affine values specially:

$$\mathcal{N}[\![V(\lambda_1(x^+).\, M)]\!] \triangleq \nu y.\, y \coloneqq_1 \lambda(x^+).\, \mathcal{N}[\![M]\!] \,\mathbf{in}\, \mathcal{N}[\![V(y)]\!]$$

Finally, the π-calculus translation $\mathcal{P}$ should use an unreplicated process to simulate ephemeral assignment:

$$\mathcal{P}[\![x \coloneqq_1 \lambda(y^+).\, M \,\mathbf{in}\, N]\!] \triangleq \mathcal{P}[\![N]\!] \mid x(y^+).\, \mathcal{P}[\![M]\!]$$

$$
\begin{aligned}
\mathcal{C}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, k(\lambda(x,k').\, \mathcal{C}[\![M]\!]k') \\
\mathcal{C}[\![MN]\!] &\triangleq \lambda k.\, \mathcal{C}[\![M]\!](\lambda v.\, \nu x. \\
&\qquad x :=_1 \big(\lambda k'.\, \mathcal{C}[\![N]\!](\lambda w.\, x := \lambda k''.\, k''w \,\mathbf{in}\, k'v)\big) \,\mathbf{in}\, v(x,k)) \\
\mathcal{C}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!] &= \lambda k.\, \nu x.\, x :=_1 (\lambda k'.\, \mathcal{C}[\![L]\!](\lambda w.\, x := \lambda k''.\, k''w \,\mathbf{in}\, kw)) \,\mathbf{in}\, \mathcal{C}[\![M]\!]k
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{vn}[\![x]\!] &\triangleq \lambda k.\, xk \\
\mathcal{C}_{vn}[\![\lambda x.\, M]\!] &\triangleq \lambda k.\, \nu f.\, f := \lambda(x,k').\, \mathcal{C}_{vn}[\![M]\!]k' \,\mathbf{in}\, kf \\
\mathcal{C}_{vn}[\![MN]\!] &\triangleq \lambda k.\, \nu h.\, h :=_1 \lambda v.\, \nu x.\, x :=_1 \begin{pmatrix} \lambda k'.\, \nu h'.\, h' :=_1 \\ \quad \lambda v.\, x := \lambda k''.\, k''v \,\mathbf{in}\, k'v \\ \mathbf{in}\, \mathcal{C}_{vn}[\![N]\!]h' \end{pmatrix} \mathbf{in}\, v(x,k) \\
&\qquad \mathbf{in}\, \mathcal{C}_{vn}[\![M]\!]h \\
\mathcal{C}_{vn}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!] &\triangleq \lambda k.\, \nu x.\, x :=_1 \begin{pmatrix} \lambda k.\, \nu h'.\, h' :=_1 \\ \quad \lambda v.\, x := \lambda k.\, kv \,\mathbf{in}\, kv \\ \mathbf{in}\, \mathcal{C}_{vn}[\![L]\!]h' \end{pmatrix} \mathbf{in}\, \mathcal{C}_{vn}[\![M]\!]k
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![x]\!]_k &\triangleq \overline{x}\langle k\rangle \\
\mathcal{E}[\![\lambda x.\, M]\!]_k &\triangleq \boldsymbol{\nu} f\ (\overline{k}\langle f\rangle \mid !f(x,k).\, \mathcal{E}[\![M]\!]_k) \\
\mathcal{E}[\![MN]\!]_k &\triangleq \boldsymbol{\nu} h\ (\mathcal{E}[\![M]\!]_h \mid h(v).\, \boldsymbol{\nu} x\ (\overline{v}\langle x,k\rangle \mid \\
&\qquad x(k').\, \boldsymbol{\nu} h'\ (\mathcal{E}[\![N]\!]_{h'} \mid h'(w).\, (\overline{k'}\langle w\rangle \mid !x(k'').\, \overline{k''}\langle w\rangle))))) \\
\mathcal{E}[\![\mathbf{let}\, x = L \,\mathbf{in}\, M]\!]_k &\triangleq \boldsymbol{\nu} x\ (\mathcal{E}[\![M]\!]_k \mid x(k').\, \boldsymbol{\nu} h'\ (\mathcal{E}[\![L]\!]_{h'} \mid h'(w).\, (\overline{k'}\langle w\rangle \mid !x(k'').\, \overline{k''}\langle w\rangle))))
\end{aligned}
$$

FIGURE 37. The CPS transform, named CPS transform, and π-encoding for call-by-need.

Putting these transforms together (Fig. 37), we arrive at the same call-by-need π-encoding found in the literature (Brock & Ostheimer, 1995; Sangiorgi & Walker, 2003).

Correctness

Now we establish the correctness of the call-by-need CPS transform $\mathcal{C}$. We do so by further decomposing it into three steps: A switch to a call-by-need calculus with rules that act *at a distance*; an annotation step; and simulation proofs for the CPS transform on annotated terms.

$$\text{Binding Contexts:} \quad B ::= [] \bigm| \mathbf{let}\, x = M \,\mathbf{in}\, B$$

$$\begin{array}{rll} B[\lambda x.\, M]N & \rightarrow B[\mathbf{let}\, x = N \,\mathbf{in}\, M] & \beta_d \\ \mathbf{let}\, x = B[V] \,\mathbf{in}\, E[x] & \rightarrow B[\mathbf{let}\, x = V \,\mathbf{in}\, E[V]] & \mathit{deref}_d \end{array}$$

FIGURE 38. Reductions for the distance call-by-need λ-calculus, λ^d_{need}.

Distance Rules

As presented, λ_{need} (Fig. 34) has certain reductions—the *lift* and *assoc* rules—that hardly seem to perform any computation. They only shuffle bindings around in preparation for a β_{need}- or *deref*-reduction, respectively. In fact, if $\mathcal{C}[\![M]\!]$ is always an administrative λ-abstraction[7], then *lift*- and *assoc*-reductions are simulated as administrative reductions alone. In a sense, the *lift* and *assoc* rules *are* administrative: They only serve to bring the parts of a redex together.

We can avoid administrative work in the source calculus by using a suggestion of Accattoli (Accattoli, 2013) for the π-calculus: We express λ_{need} using rules that apply *at a distance*, that is, where parts of a redex are separated by an evaluation context.[8] The new calculus, λ^d_{need}, supplants the fine-grained *lift* and *assoc* rules with coarser β and *deref* rules. The syntax is the same as for λ_{need} (Fig. 34), except that we specify that some evaluation contexts are *binding contexts*; the reductions are given in Fig. 38.

[7]We could always not mark these as administrative, but then we would lose the flexibility that administrative congruence provides.

[8]Of course, the *deref* rule already works this way in part.
An alternative is suggested by Chang and Felleisen (Chang & Felleisen, 2012), who use distance rules and manage to do away with the *deref* rule as well, but at the cost of a more complex β-rule.

Proposition 40. *An λ_{need} term reduces to an answer, diverges, or gets stuck by the distance rules if and only if it does so by the original rules.*

Proof. Since $\mapsto_{\beta_d}$ is the same as $\mapsto^{\star}_{\mathit{lift}}\mapsto_{\beta_{need}}$, $\mapsto_{deref_d}$ is the same as $\mapsto^{\star}_{assoc}\mapsto_{deref}$, and the language is deterministic, it suffices to define a backward simulation that compares terms up to *lift* and *assoc*. □

Annotations

The single **let** construct does not tell the full story of a standard reduction sequence in λ_{need}: There is an implicit statefulness that is made manifest by the CPS transform. Specifically, there are three stages in the life cycle of a **let** binding:

Suspended Initially, the binding $\mathbf{let}\, x \;=\; M\, \mathbf{in}\, N$ represents a *suspended* computation. Computation takes place within N.

Active For x to be demanded, N must reduce to the form $E[x]$. Then the binding becomes *active*, with the form $\mathbf{let}\, x = M\, \mathbf{in}\, E[x]$, and computation takes place within M.

Memoized Eventually, M becomes an answer $B[V]$, and the body $E[x]$ receives V while the bindings in B are added to the environment. Subsequently, the binding is $\mathbf{let}\, x = V\, \mathbf{in}\, N$, and computation takes place within N.

The CPS translation exposes this state, which makes it difficult to relate a term to its CPS form: In what state is $\mathbf{let}\, x = V\, \mathbf{in}\, E[x]$? In fact, it could be in *any* of the three states, and thus the running CPS program could have any of three forms.

Therefore we annotate each **let**, giving it a subscript **s**, **a**, or **v** (for *suspended*, *active*, or *value*, respectively). We will need new reduction rules,

$$
\begin{array}{rrl}
\text{Expressions:} & M, N ::= & x \mid V \mid M N \\
& & \mid \mathbf{let}_s\, x = M \,\mathbf{in}\, N \\
& & \mid \mathbf{let}_a\, x = M \,\mathbf{in}\, E[x] \\
& & \mid \mathbf{let}_v\, x = V \,\mathbf{in}\, N \\
\text{Evaluation Contexts:} & E, F ::= & \square \mid E M \\
& & \mid \mathbf{let}_s\, x = M \,\mathbf{in}\, E \\
& & \mid \mathbf{let}_a\, x = E \,\mathbf{in}\, F[x] \\
& & \mid \mathbf{let}_v\, x = V \,\mathbf{in}\, E \\
\text{Binding Contexts:} & B ::= & \square \mid \mathbf{let}_s\, x = M \,\mathbf{in}\, B \\
& & \mid \mathbf{let}_v\, x = V \,\mathbf{in}\, B
\end{array}
$$

$$
\begin{array}{rll}
B[\lambda x.\, M]N \to & B[\mathbf{let}_s\, x = N \,\mathbf{in}\, M] & \beta_a \\
\mathbf{let}_s\, x = M \,\mathbf{in}\, E[x] \to & \mathbf{let}_a\, x = M \,\mathbf{in}\, E[x] & \mathit{act} \\
\mathbf{let}_a\, x = B[V] \,\mathbf{in}\, E[x] \to & B[\mathbf{let}_v\, x = V \,\mathbf{in}\, E[V]] & \mathit{deact} \\
\mathbf{let}_v\, x = V \,\mathbf{in}\, E[x] \to & \mathbf{let}_v\, x = V \,\mathbf{in}\, E[V] & \mathit{deref}_a
\end{array}
$$

FIGURE 39. The annotated call-by-need λ-calculus, $\lambda^a_{\mathit{need}}$.

which we call *act* and *deact*, to represent binding state transitions; from the perspective of $\lambda^d_{\mathit{need}}$, these will be administrative. See Fig. 39 for the resulting calculus $\lambda^a_{\mathit{need}}$. We write $\mathcal{A}[\![M]\!]$ for the annotation of a $\lambda^d_{\mathit{need}}$ term M, which consists simply of tagging each **let** with **s**.

Our $\lambda^a_{\mathit{need}}$ is much like previous languages with similar goals. In particular, both Brock and Ostheimer (Brock & Ostheimer, 1995) and Danvy and Zerny (Danvy & Zerny, 2013) include versions of what we call $\mathbf{let_a}$, the "active **let**."

Proposition 41.

1. *$M \Downarrow$ if and only if $\mathcal{A}[\![M]\!] \Downarrow$.*

2. *$M \Uparrow$ if and only if $\mathcal{A}[\![M]\!] \Uparrow$.*

3. *$M \not\Downarrow$ if and only if $\mathcal{A}[\![M]\!] \not\Downarrow$.*

Annotated CPS Transform

Now we must show that the call-by-need CPS transform, as a function from λ^{a}_{need} to $\lambda^{:=_1}_{cps}$, is correct. First, we need to extend the CPS transform to the annotated terms. See Fig. 40, where we have also marked which λ-abstractions are administrative.

Most of the first part of $\mathcal{C}_a$ is unsurprising: A $\mathbf{let_s}$ translates as a suspended computation, as appears in $\mathcal{C}_a[\![MN]\!]$. A $\mathbf{let_v}$ translates as a memo-thunk. However, $\mathbf{let_a}$ is a challenge. To see why, consider a suspended computation:

$$\mathcal{C}_a[\![\mathbf{let}_s\, x = M \,\mathbf{in}\, N]\!]K =_{ad} \nu x.\, x :=_1 \bar{\lambda} k.\ \cdots\ \mathbf{in}\, \mathcal{C}_a[\![N]\!]K$$

The computation of x is suspended, pending its need. Then, x will be needed when N reduces to a term of the form $E[x]$. At that point in the computation, we expect $\mathcal{C}_a[\![E[x]]\!]$ to have evaluated to some term $E'[xK']$, where E' is a CPS evaluation context and K' is a continuation.

$$\mathcal{C}_a[\![\mathbf{let}_s\, x = M \,\mathbf{in}\, E[x]]\!]K =_{ad} \nu x.\, x :=_1 \bar{\lambda} k.\ \cdots\ \mathbf{in}\, E'[xK']$$

What happens next is that the *deref*$_1$ rule fires:

$$\nu x.\, x :=_1 \bar{\lambda} k.\ \cdots\ \mathbf{in}\, E'[xK'] \mapsto \nu x.\, E'[(\bar{\lambda} k.\ \cdots)K']$$

Since this *deref*$_1$-reduction is what activates the computation of x, we expect that $\nu x.\, E'[(\bar{\lambda} k.\ \cdots)K']$ should be the shape of the CPS term corresponding to an active **let**. But this means we must be able to translate *evaluation contexts* as well as terms. For the uniform CPS transform, we were content to have a

$$
\begin{aligned}
\mathcal{C}_a\llbracket x\rrbracket &\triangleq \bar\lambda k.\, xk\\
\mathcal{C}_a\llbracket \lambda x.\, M\rrbracket &\triangleq \bar\lambda k.\, k(\bar\lambda(x,k').\, \mathcal{C}_a\llbracket M\rrbracket k')\\
\mathcal{C}_a\llbracket MN\rrbracket &\triangleq \bar\lambda k.\, \mathcal{C}_a\llbracket M\rrbracket(\lambda v.\, \nu x.\\
&\qquad x :=_1 \bar\lambda k'.\, \mathcal{C}_a\llbracket N\rrbracket(\lambda v.\, x := \lambda k''.\, k''v \,\mathbf{in}\, k'v)\\
&\qquad \mathbf{in}\, v(x,k))\\
\mathcal{C}_a\llbracket \mathbf{let}_s\, x = M \,\mathbf{in}\, N\rrbracket &\\
&\triangleq \bar\lambda k.\, \nu x.\, x :=_1 \bar\lambda k'.\, \mathcal{C}_a\llbracket M\rrbracket(\lambda v.\, x := \bar\lambda k''.\, k''v \,\mathbf{in}\, k'v)\\
&\qquad \mathbf{in}\, \mathcal{C}_a\llbracket N\rrbracket k\\
\mathcal{C}_a\llbracket \mathbf{let}_a\, x = M \,\mathbf{in}\, E[x]\rrbracket &\\
&\triangleq \bar\lambda k.\, \nu x.\, \mathcal{C}_a\llparenthesis E\rrparenthesis[\bar\lambda k'.\, \mathcal{C}_a\llbracket M\rrbracket(\lambda v.\, x := \bar\lambda k''.\, k''v \,\mathbf{in}\, k'v)]k\\
\mathcal{C}_a\llbracket \mathbf{let}_v\, x = V \,\mathbf{in}\, N\rrbracket &\\
&\triangleq \bar\lambda k.\, \nu x.\, x := \mathcal{C}_a\llbracket V\rrbracket \,\mathbf{in}\, \mathcal{C}_a\llbracket N\rrbracket k
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_a\llparenthesis \Box\rrparenthesis &\triangleq \Box\\
\mathcal{C}_a\llparenthesis EN\rrparenthesis &\triangleq \bar\lambda k.\, \mathcal{C}_a\llparenthesis E\rrparenthesis(\lambda v.\, \nu x.\\
&\qquad x :=_1 \bar\lambda k'.\, \mathcal{C}_a\llbracket N\rrbracket(\lambda v.\, x := \bar\lambda k''.\, k''v \,\mathbf{in}\, k'v)\\
&\qquad \mathbf{in}\, v(x,k))\\
\mathcal{C}_a\llparenthesis \mathbf{let}_s\, x = M \,\mathbf{in}\, E\rrparenthesis &\\
&\triangleq \bar\lambda k.\, \nu x.\, x :=_1 \bar\lambda k'.\, \mathcal{C}_a\llbracket M\rrbracket(\lambda v.\, x := \bar\lambda k''.\, k''v \,\mathbf{in}\, k'v)\\
&\qquad \mathbf{in}\, \mathcal{C}_a\llparenthesis E\rrparenthesis k\\
\mathcal{C}_a\llparenthesis \mathbf{let}_a\, x = E \,\mathbf{in}\, F[x]\rrparenthesis &\\
&\triangleq \bar\lambda k.\, \nu x.\, \mathcal{C}_a\llparenthesis F\rrparenthesis[\bar\lambda k'.\, \mathcal{C}_a\llparenthesis E\rrparenthesis(\lambda v.\, x := \bar\lambda k''.\, k''v \,\mathbf{in}\, k'v)]k\\
\mathcal{C}_a\llparenthesis \mathbf{let}_v\, x = V \,\mathbf{in}\, E\rrparenthesis &\\
&\triangleq \bar\lambda k.\, \nu x.\, x := \mathcal{C}_a\llbracket V\rrbracket \,\mathbf{in}\, \mathcal{C}_a\llparenthesis E\rrparenthesis k
\end{aligned}
$$

FIGURE 40. The call-by-need CPS transform on annotated terms and contexts, $\mathcal{C}_a$.

lemma (Proposition 19) that merely *asserted* that there was some K that served to translate a source evaluation context. Now that contexts are a crucial part of the CPS transform, we need to make the translation explicit. Hence the context part of the CPS transform, which we write using round brackets, like $\mathcal{C}_a(\!|E|\!)$. Its definition is easily derived from that of $\mathcal{C}_a$, so that we have:

Proposition 42. $\mathcal{C}_a[\![E[M]]\!] \equiv \mathcal{C}_a(\!|E|\!)[\mathcal{C}_a[\![M]\!]]$

Proof. An easy induction on E. $\square$

Again we define an administrative congruence relation. We will want to be able to rearrange bindings when it is safe to do so; accordingly, we adopt a *lift* rule, a generalization of the *lift* rule from call-by-need:

$$\begin{array}{rll} E[E'[C]] \rightarrow_{ad} E'[E[C]] & & \textit{lift} \\ (\bar{\lambda}(x^+).\, M)(V^+) \rightarrow_{ad} M\{V^+/x^+\} & & \beta_{ad} \end{array}$$

As usual, the transitive closure of $\rightarrow_{ad}$ is $\rightarrow^+_{ad}$, and its reflexive and transitive closure is $\rightarrow^\star_{ad}$. Its reflexive, symmetric, and transitive closure, restricted[9] to safe terms, is $=_{ad}$.

We will also use $\mapsto_{ad}$ to refer to an invocation of β_{ad} at the top level, which we call an *administrative standard reduction.* (A *lift*-reduction is never proper, as it is never necessary.) As before, an *administrative answer* is a term that cannot take an administrative standard reduction.

The confluence, standardization, and commutativity results we need are similar to before:

Proposition 43. *The relation* $=_{ad}$

[9]The restriction is necessary because safety is not closed backward, i.e.there are unsafe terms that reduce to safe ones.

1. *is* confluent*, so that if $M =_{ad} M'$, then there is some N such that $M \twoheadrightarrow^{\star}_{ad} N$ and $M' \twoheadrightarrow^{\star}_{ad} N$; and*

2. *has the* standardization *property, so that if $M \twoheadrightarrow^{\star}_{ad} N$ and N is an administrative answer, then there is an administrative answer M' such that $M \mapsto^{\star}_{ad} M' \twoheadrightarrow^{\star}_{ad} N$.*

Proof.

1. Because *lift* and β_{ad} do not overlap and they are both left-linear, we can use modularity (Appel, van Oostrom, & Simonsen, 2010) to prove confluence from the confluence of each rule separately. The *lift* rule is symmetric and thus trivially confluent. The β_{ad} rule is simply the β rule restricted to a subcalculus, so it is also confluent.

2. Since *lift* and β_{ad} trivially commute (they do not even interact), we can perform the *lift* steps last, giving $M \twoheadrightarrow^{\star}_{\beta_{ad}} N' \twoheadrightarrow^{\star}_{lift} N$. Since $\twoheadrightarrow^{\star}_{lift}$ cannot create a proper standard reduction, N' must also be an administrative answer; hence standardization of β-reduction applies, completing the proof. □

Proposition 44.

1. *If $M \twoheadrightarrow^{\star}_{ad} M' \mapsto_{pr} N$ and M is an administrative answer, then there is N' with $M \mapsto_{pr} N' \twoheadrightarrow^{\star}_{ad} N$.*

2. *If $M \twoheadleftarrow^{\star}_{ad} M' \mapsto_{pr} N$, then there is N' with $M \mapsto_{pr} N' \twoheadleftarrow^{\star}_{ad} N$.*

Proof. The *lift* rule can neither create nor destroy *any* redexes, and internal reductions still cannot destroy a standard redex, so the proof is essentially the same as that of Proposition 17. □

Lemma 45. *If $M =_{ad}\mapsto_{pr} N$, then $M \mapsto^{+}_{pr^1} =_{ad} N$.*

Proof. The same as the proof of Lemma 18, this time using Propositions 43 and 44. $\square$

For the uniform CPS transform, we characterized the action of translation on contexts as Proposition 19: $\mathcal{C}_a[\![E[M]]\!]K$ will reduce to $\mathcal{C}[\![M]\!]K'$, where K' is some continuation that represents E. This case will be more complex, however: In λ_{need}, an evaluation context contains both bindings *and* work to be done.[10] A continuation alone only captures the latter. Therefore, for call-by-need, we will need to translate the bindings as well. Our approach is to split context translation into a function $\mathcal{B}$ providing a CPS binding context and a function $\mathcal{K}$ providing a continuation (in which the bindings from $\mathcal{B}$ are in scope). Putting them together, we will be able to relate the original context transform $\mathcal{C}_a(\!|-|\!)$ to the split transform.

$$\begin{aligned}
\mathcal{B}(\!|\square|\!) &\triangleq [] \\
\mathcal{B}(\!|E\,M|\!) &\triangleq \mathcal{B}(\!|E|\!) \\
\mathcal{B}(\!|\mathbf{let}_s\, x = M \,\mathbf{in}\, E|\!) &\triangleq \nu x.\, x :=_1 \\
&\qquad\qquad \bar{\lambda} k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x := \bar{\lambda} k''.\, k'' v \,\mathbf{in}\, k' v) \\
&\qquad \mathbf{in}\, \mathcal{B}(\!|E|\!) \\
\mathcal{B}(\!|\mathbf{let}_a\, x = E \,\mathbf{in}\, F[x]|\!) &\triangleq \nu x.\, \mathcal{B}(\!|F|\!)[\mathcal{B}(\!|E|\!)] \\
\mathcal{B}(\!|\mathbf{let}_v\, x = V \,\mathbf{in}\, E|\!) &\triangleq \nu x.\, x := \mathcal{C}_a[\![V]\!] \,\mathbf{in}\, \mathcal{B}(\!|E|\!)
\end{aligned}$$

[10]Danvy and Zerny (Danvy & Zerny, 2013) make a similar observation about call-by-need evaluation contexts; they then derive a call-by-need language that separates the two parts.

$$
\begin{aligned}
\mathcal{K}(\!|\square|\!) : K &\triangleq K \\
\mathcal{K}(\!|EM|\!) : K &\triangleq \mathcal{K}(\!|E|\!) : \lambda v.\, \nu x.\, x \coloneqq_1 \\
&\qquad\qquad \bar{\lambda} k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x \coloneqq \bar{\lambda} k''.\, k'' v \,\mathbf{in}\, k' v) \\
&\qquad\quad \mathbf{in}\, v(x, K) \\
\mathcal{K}(\!|\mathbf{let}_s\, x = M \,\mathbf{in}\, E|\!) : K &\triangleq \mathcal{K}(\!|E|\!) : K \\
\mathcal{K}(\!|\mathbf{let}_a\, x = E \,\mathbf{in}\, F[x]|\!) : K &\triangleq \mathcal{K}(\!|E|\!) : \lambda v.\, x \coloneqq \bar{\lambda} k''.\, k'' v \,\mathbf{in}\, (\mathcal{K}(\!|F|\!) : K) v \\
\mathcal{K}(\!|\mathbf{let}_v\, x = V \,\mathbf{in}\, E|\!) : K &\triangleq \mathcal{K}(\!|E|\!) : K
\end{aligned}
$$

The following proposition shows the relationship between $\mathcal{C}_a$, $\mathcal{B}$, and $\mathcal{K}$. Note that, like Proposition 19, it demonstrates the action of administrative reductions: In this case, they serve to bring the continuation inward while preserving the bindings in the context. We use T here to denote a value that is, in particular, a thunk of the form $\lambda k.\, M$.

Proposition 46. $\mathcal{C}_a(\!|E|\!)[T]K \equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]$

Proof. By induction on E. First, some shorthand will clarify:

$$\mathbf{let}_\ell\, x = V \,\mathbf{in}\, P \triangleq \nu x.\, x \coloneqq_1 \bar{\lambda} k.\, V(\lambda v.\, x \coloneqq \bar{\lambda} k.\, kv \,\mathbf{in}\, kv) \,\mathbf{in}\, P$$

Now:

– For $E \equiv \Box$:

$$\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv \Box[T]K \\
&\triangleq TK \\
&\triangleq \Box[TK] \\
&\triangleq \mathcal{B}(\!|\Box|\!)[T(\mathcal{K}(\!|\Box|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]
\end{aligned}$$

– For $E \equiv E'M$:

$$\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv \mathcal{C}_a(\!|E'M|\!)[T]K \\
&\triangleq (\bar{\lambda}k.\, \mathcal{C}_a(\!|E'|\!)(\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\, \mathbf{in}\, v(x,k)))[T]K \\
&\equiv (\bar{\lambda}k.\, \mathcal{C}_a(\!|E'[T]|\!)(\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\, \mathbf{in}\, v(x,k)))K \\
&\mapsto_{ad} \mathcal{C}_a(\!|E'[T]|\!)(\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\, \mathbf{in}\, v(x,K)) \\
&\mapsto_{ad}^{\star} \mathcal{B}(\!|E'|\!)[T(\mathcal{K}(\!|E'|\!) : \lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\, \mathbf{in}\, v(x,K))] \quad \text{(by I.H.)} \\
&\triangleq \mathcal{B}(\!|E'M|\!)[T(\mathcal{K}(\!|E'M|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]
\end{aligned}$$

– For $E \equiv \mathbf{let}_s\, x = M\, \mathbf{in}\, E'$:

$$
\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv \mathcal{C}_a(\!|\mathbf{let}_s\, x = M\, \mathbf{in}\, E'|\!)[T]K \\
&\triangleq (\bar{\lambda}k.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\, \mathbf{in}\, \mathcal{C}_a(\!|E'|\!)k)[T]K \\
&\equiv (\bar{\lambda}k.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\, \mathbf{in}\, \mathcal{C}_a(\!|E'[T]|\!)k)K \\
&\mapsto_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\, \mathbf{in}\, \mathcal{C}_a(\!|E'[T]|\!)K \\
&\mapsto^{\star}_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!]\, \mathbf{in}\, \mathcal{B}(\!|E'|\!)[T(\mathcal{K}(\!|E'|\!) : K)k] \qquad \text{(by I.H.)} \\
&\triangleq \mathcal{B}(\!|\mathbf{let}_s\, x = M\, \mathbf{in}\, E'|\!)[T(\mathcal{K}(\!|\mathbf{let}_s\, x = M\, \mathbf{in}\, E'|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]
\end{aligned}
$$

– For $E \equiv \mathbf{let}_a\, x = E'\, \mathbf{in}\, F[x]$:

$$
\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv \mathcal{C}_a(\!|\mathbf{let}_a\, x = E'\, \mathbf{in}\, F[x]|\!)[T]K \\
&\equiv (\bar{\lambda}k.\, \nu x.\, \mathcal{C}_a(\!|F|\!)[\bar{\lambda}k'.\, \mathcal{C}_a(\!|E'|\!)(\lambda v.\, x := \bar{\lambda}k''.\, k''v\, \mathbf{in}\, k'v)]k)[T]K \\
&\equiv (\bar{\lambda}k.\, \nu x.\, \mathcal{C}_a(\!|F|\!)[\bar{\lambda}k'.\, \mathcal{C}_a(\!|E'|\!)[T](\lambda v.\, x := \bar{\lambda}k''.\, k''v\, \mathbf{in}\, k'v)]k)K \\
&\mapsto_{ad} \nu x.\, \mathcal{C}_a(\!|F|\!)[\bar{\lambda}k'.\, \mathcal{C}_a(\!|E'|\!)[T](\lambda v.\, x := \bar{\lambda}k''.\, k''v\, \mathbf{in}\, k'v)]K \\
&\mapsto^{\star}_{ad} \nu x.\, \mathcal{B}(\!|F|\!)[(\bar{\lambda}k'.\, \mathcal{C}_a(\!|E'|\!)[T](\lambda v.\, x := \bar{\lambda}k''.\, k''v\, \mathbf{in}\, (\mathcal{K}(\!|F|\!) : k')v))K] \quad \text{(by I.H.)} \\
&\mapsto_{ad} \nu x.\, \mathcal{B}(\!|F|\!)[\mathcal{C}_a(\!|E'|\!)[T](\lambda v.\, x := \bar{\lambda}k''.\, k''v\, \mathbf{in}\, (\mathcal{K}(\!|F|\!) : K)v)] \\
&\mapsto^{\star}_{ad} \nu x.\, \mathcal{B}(\!|F|\!)[\mathcal{B}(\!|E'|\!)[T(\mathcal{K}(\!|E'|\!) : \lambda v.\, x := \bar{\lambda}k''.\, k''v\, \mathbf{in}\, (\mathcal{K}(\!|F|\!) : K)v))]] \quad \text{(by I.H.)} \\
&\triangleq \nu x.\, \mathcal{B}(\!|\mathbf{let}_a\, x = E'\, \mathbf{in}\, F[x]|\!)[T(\mathcal{K}(\!|\mathbf{let}_a\, x = E'\, \mathbf{in}\, F[x]|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)]
\end{aligned}
$$

– For $E \equiv \mathbf{let}_v\, x = V \,\mathbf{in}\, E'$:

$$\begin{aligned}
\mathcal{C}_a(\!|E|\!)[T]K &\equiv \mathcal{C}_a(\!|\mathbf{let}_v\, x = V \,\mathbf{in}\, E'|\!)[T]K \\
&\triangleq (\bar{\lambda}k.\, \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \,\mathbf{in}\, \mathcal{C}_a(\!|E'|\!)k)[T]K \\
&\equiv (\bar{\lambda}k.\, \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \,\mathbf{in}\, \mathcal{C}_a(\!|E'[T]|\!)k)K \\
&\mapsto_{ad} \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \,\mathbf{in}\, \mathcal{C}_a(\!|E'[T]|\!)K \\
&\mapsto^{\star}_{ad} \nu x.\, x \coloneqq \mathcal{C}_a[\![V]\!] \,\mathbf{in}\, \mathcal{B}(\!|E'|\!)[T(\mathcal{K}(\!|E'|\!) : K)] && \text{(by I.H.)} \\
&\triangleq \mathcal{B}(\!|\mathbf{let}_v\, x = V \,\mathbf{in}\, E'|\!)[T(\mathcal{K}(\!|\mathbf{let}_v\, x = V \,\mathbf{in}\, E'|\!) : K)] \\
&\equiv \mathcal{B}(\!|E|\!)[T(\mathcal{K}(\!|E|\!) : K)] && \square
\end{aligned}$$

Now we can show how a term in a context is transformed:

Corollary 47. $\mathcal{C}_a[\![E[M]]\!]K \mapsto^{\star}_{ad} \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![M]\!](\mathcal{K}(\!|E|\!) : K)]$.

Proof. From Propositions 42 and 46. □

Not all λ_{need} contexts affect the continuation. In particular, binding contexts never alter the continuation at all. This is not surprising, since binding contexts are precisely those that do not affect the flow of control.

Proposition 48. *For any λ^a_{need} binding context B and $\lambda^{:=_1}_{cps}$ continuation K, $\mathcal{K}(\!|B|\!) : K \equiv K$.*

Proof. Trivial induction on B. Note that the clauses of $\mathcal{K}$ that do nothing but recurse are precisely those for the parts of a binding context. □

Corollary 49. *For any λ^a_{need} binding context B, λ^a_{need} term M, and $\lambda^{:=_1}_{cps}$ continuation K, $\mathcal{C}_a[\![B[M]]\!]K \mapsto^{\star}_{ad} \mathcal{B}(\!|B|\!)[\mathcal{C}_a[\![M]\!]K]$.*

Proof. Immediate from Corollary 47 and Proposition 48. □

Now we turn to correctness. Our forward simulation follows the same construction as that of the uniform CPS transform:

Definition 50. *For a λ^{a}_{need} term M and $\lambda^{:=1}_{cps}$ term P, let $M \sim P$ when $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} =_{ad} P$.*

To prove that $\sim$ is a forward simulation, we go through the diagrams in Eq. (4.6) once again. The first, third, and fourth are exactly as before:

Lemma 51. $M \sim \mathcal{C}_a[\![M]\!]\,\mathbf{ret}$. □

Lemma 52. *If $M \sim P$ and AM, then $P \Downarrow$.* □

Lemma 53. *If $M \sim P$ and sM, then $P \not\Downarrow$.* □

The second diagram can be proved using much the same strategy as for Lemma 29, but the more complex source and target languages make the calculations heavier. To review, the steps we take to prove the simulation are:

1. Show that, if $M \mapsto N$ by a reduction in the empty context, then we have $\mathcal{C}_a[\![M]\!]K \mapsto^{\star}=_{ad} \mathcal{C}_a[\![N]\!]K$.
2. Allow the reduction to occur in any evaluation context, not only at the top of the term.
3. Let the CPS term be *any* $P =_{ad} \mathcal{C}_a[\![]\!]M$.

Now we begin with step one immediately:

Lemma 54. *If $M \mapsto N$ by a reduction in the empty context, then $\mathcal{C}_a[\![M]\!]K \mapsto^{+}_{pr1}=_{ad} \mathcal{C}_a[\![N]\!]K$.*

Proof. This time we have four reduction rules, so there are four cases.

– For a β-reduction:

$$\mathcal{C}_a[\![B[\lambda x.\, M]N]\!]K$$

$$\triangleq (\bar{\lambda}k.\,(\mathcal{C}_a[\![B[\lambda x.\, M]]\!](\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}\, v(x,k)))K$$

$$\mapsto_{ad} \mathcal{C}_a[\![B[\lambda x.\, M]]\!](\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}\, v(x,K))$$

$$\mapsto_{ad}^{\star} \mathcal{B}(\!|B|\!)[\mathcal{C}_a[\![\lambda x.\, M]\!](\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}\, v(x,K))] \qquad \text{(by Corollary 49)}$$

$$\triangleq \mathcal{B}(\!|B|\!)[(\bar{\lambda}k.\, k(\bar{\lambda}(x,k').\, \mathcal{C}_a[\![M]\!]k'))(\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}\, v(x,K))]$$

$$\mapsto_{ad} \mathcal{B}(\!|B|\!)[(\lambda v.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}\, v(x,K))(\bar{\lambda}(x,k').\, \mathcal{C}_a[\![M]\!]k')]$$

$$\mapsto_{pr} \mathcal{B}(\!|B|\!)[\mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}(\bar{\lambda}(x,k').\, \mathcal{C}_a[\![M]\!]k')(x,K)]$$

$$\mapsto_{ad} \mathcal{B}(\!|B|\!)[\mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}\, \mathcal{C}_a[\![M]\!]K]$$

$$\leftarrow\!\!\!\dashv_{ad} \mathcal{B}(\!|B|\!)[(\bar{\lambda}k.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}\, \mathcal{C}_a[\![M]\!]k)K]$$

$$\leftarrow\!\!\!\dashv_{ad}^{\star} \mathcal{C}_a(\!|B|\!)[\mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}\, \mathcal{C}_a[\![M]\!]]K \qquad \text{(by Corollary 49)}$$

$$\leftarrow\!\!\!\dashv_{ad} (\bar{\lambda}k.\, \mathcal{C}_a(\!|B|\!)[\mathbf{let}_\ell\, x = \mathcal{C}_a[\![N]\!]\,\mathbf{in}\, \mathcal{C}_a[\![M]\!]]k)K$$

$$\triangleq \mathcal{C}_a[\![B[\mathbf{let}_s\, x = N\,\mathbf{in}\, M]]\!]K$$

– For an *act*-reduction:

$$
\begin{aligned}
&\mathcal{C}_a[\![\mathbf{let}_s\, x = M \,\mathbf{in}\, E[x]]\!]K\\
&\quad\triangleq (\bar{\lambda}k.\ \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, \mathcal{C}_a[\![E[x]]\!])K\\
&\quad\mapsto_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, \mathcal{C}_a[\![E[x]]\!]K\\
&\quad\mapsto^{\star}_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![x]\!](\mathcal{K}(\!|E|\!) : K)] && \text{(by Corollary 47)}\\
&\quad\triangleq \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, \mathcal{B}(\!|E|\!)[(\bar{\lambda}k.\, xk)(\mathcal{K}(\!|E|\!) : K)]\\
&\quad\mapsto_{ad} \mathbf{let}_\ell\, x = \mathcal{C}_a[\![M]\!] \,\mathbf{in}\, \mathcal{B}(\!|E|\!)[x(\mathcal{K}(\!|E|\!) : K)]\\
&\quad\triangleq \nu x.\, x \coloneqq_1 \bar{\lambda}k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \,\mathbf{in}\, k'v) \,\mathbf{in}\, \mathcal{B}(\!|E|\!)[x(\mathcal{K}(\!|E|\!) : K)]\\
&\quad\mapsto_{pr} \nu x.\, \mathcal{B}(\!|E|\!)[(\bar{\lambda}k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \,\mathbf{in}\, k'v))(\mathcal{K}(\!|E|\!) : K)]\\
&\quad\leftarrow^{\star}_{ad} (\bar{\lambda}k.\, \nu x.\, \mathcal{C}_a(\!|E|\!)[\bar{\lambda}k'.\, \mathcal{C}_a[\![M]\!](\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \,\mathbf{in}\, k'v)])K && \text{(by Corollary 47)}\\
&\quad\triangleq \mathcal{C}_a[\![\mathbf{let}_a\, x = M \,\mathbf{in}\, E[x]]\!]K
\end{aligned}
$$

– For a *deact*-reduction:

$$
\begin{aligned}
&\mathcal{C}_a[\![\mathbf{let}_a\, x = B[V] \,\mathbf{in}\, E[x]]\!]K\\
&\quad\triangleq (\bar{\lambda}k.\, \nu x.\, \mathcal{C}_a(\!|E|\!)[\bar{\lambda}k'.\, \mathcal{C}_a[\![B[V]]\!](\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \,\mathbf{in}\, k'v)])K\\
&\quad\mapsto_{ad} \nu x.\, \mathcal{C}_a(\!|E|\!)[\bar{\lambda}k'.\, \mathcal{C}_a[\![B[V]]\!](\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \,\mathbf{in}\, k'v)]K\\
&\quad\mapsto^{\star}_{ad} \nu x.\, \mathcal{B}(\!|E|\!)[(\bar{\lambda}k'.\, \mathcal{C}_a[\![B[V]]\!](\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \,\mathbf{in}\, k'v))(\mathcal{K}(\!|E|\!) : K)] && \text{(by Proposition 46)}\\
&\quad\mapsto_{ad} \nu x.\, \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![B[V]]\!](\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \,\mathbf{in}\, (\mathcal{K}(\!|E|\!) : K)v)]\\
&\quad\mapsto^{\star}_{ad} \nu x.\, \mathcal{B}(\!|E|\!)[\mathcal{B}(\!|B|\!)[\mathcal{C}_a[\![V]\!](\lambda v.\, x \coloneqq \bar{\lambda}k''.\, k''v \,\mathbf{in}\, (\mathcal{K}(\!|E|\!) : K)v)]] && \text{(by Corollary 49)}
\end{aligned}
$$

Letting $\bar{\lambda}k.\,kW \triangleq \mathcal{C}_a[\![V]\!]$:

$$\triangleq \nu x.\,\mathcal{B}(\!|E|\!)[\mathcal{B}(\!|B|\!)[(\bar{\lambda}k.\,kW)(\lambda v.\,x \coloneqq \bar{\lambda}k''.\,k''v \;\mathbf{in}\; (\mathcal{K}(\!|E|\!) : K)v)]]$$

$$\mapsto_{ad} \nu x.\,\mathcal{B}(\!|E|\!)[\mathcal{B}(\!|B|\!)[(\lambda v.\,x \coloneqq \bar{\lambda}k''.\,k''v \;\mathbf{in}\; (\mathcal{K}(\!|E|\!) : K)v)W]]$$

$$\mapsto_{pr} \nu x.\,\mathcal{B}(\!|E|\!)[\mathcal{B}(\!|B|\!)[x \coloneqq \bar{\lambda}k''.\,k''W \;\mathbf{in}\; (\mathcal{K}(\!|E|\!) : K)W]]$$

$$\to_{ad} \mathcal{B}(\!|B|\!)[\nu x.\,\mathcal{B}(\!|E|\!)[x \coloneqq \bar{\lambda}k''.\,k''W \;\mathbf{in}\; (\mathcal{K}(\!|E|\!) : K)W]]$$

$$\to_{ad} \mathcal{B}(\!|B|\!)[\nu x.\,x \coloneqq \bar{\lambda}k''.\,k''W \;\mathbf{in}\; \mathcal{B}(\!|E|\!)[(\mathcal{K}(\!|E|\!) : K)W]]$$

$$\leftarrow\!\!\dashv_{ad} \mathcal{B}(\!|B|\!)[\nu x.\,x \coloneqq \bar{\lambda}k''.\,k''W \;\mathbf{in}\; \mathcal{B}(\!|E|\!)[(\bar{\lambda}k.\,kW)(\mathcal{K}(\!|E|\!) : K)]]$$

$$\triangleq \mathcal{B}(\!|B|\!)[\nu x.\,x \coloneqq \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![V]\!](\mathcal{K}(\!|E|\!) : K)]]$$

$$\leftarrow\!\!\dashv^{\star}_{ad} \mathcal{B}(\!|B|\!)[\nu x.\,x \coloneqq \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{C}_a[\![E[V]]\!]K] \qquad \text{(by Corollary 47)}$$

$$\leftarrow\!\!\dashv^{\star}_{ad} \mathcal{C}_a[\![B[\mathbf{let}_v\, x = V \;\mathbf{in}\; E[V]]]\!]K \qquad \text{(by Corollary 49)}$$

– For a *deref*-reduction:

$$\mathcal{C}_a[\![\mathbf{let}_v\, x = V \;\mathbf{in}\; E[x]]\!]K$$

$$\triangleq \bar{\lambda}k.\,(\nu x.\,x \coloneqq \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{C}_a[\![E[x]]\!]k)K$$

$$\mapsto_{ad} \nu x.\,x \coloneqq \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{C}_a[\![E[x]]\!]K$$

$$\mapsto^{\star}_{ad} \nu x.\,x \coloneqq \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{B}(\!|E|\!)[x(\mathcal{K}(\!|E|\!) : K)] \qquad \text{(by Corollary 47)}$$

$$\mapsto_{pr} \nu x.\,x \coloneqq \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![V]\!](\mathcal{K}(\!|E|\!) : K)]$$

$$\leftarrow\!\!\dashv^{\star}_{ad} \nu x.\,x \coloneqq \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{C}_a[\![E[V]]\!]K \qquad \text{(by Corollary 47)}$$

$$\leftarrow\!\!\dashv_{ad} (\bar{\lambda}k.\,\nu x.\,x \coloneqq \mathcal{C}_a[\![V]\!] \;\mathbf{in}\; \mathcal{C}_a[\![E[V]]\!]k)K$$

$$\triangleq \mathcal{C}_a[\![\mathbf{let}_v\, x = V \;\mathbf{in}\; E[V]]\!]K \qquad \square$$

Now we proceed to the second step, which allows the reduction to take place in a larger context:

Proposition 55. *If $M \mapsto N$, then $\mathcal{C}_a[\![M]\!]K \mapsto^{+}_{pr1} =_{ad} \mathcal{C}_a[\![N]\!]K$.*

Proof. As before, by definition of $\mapsto$, we have $M \equiv E[M']$ and $N \equiv E[N']$, where $M' \mapsto N'$ at top level.

$$\begin{aligned}
\mathcal{C}_a[\![M]\!]K &\equiv \mathcal{C}_a[\![E[M']]\!]K \\
&\mapsto^{\star}_{ad} \mathcal{B}(\!|E|\!)[\mathcal{C}_a[\![M']\!](\mathcal{K}(\!|E|\!) : K)] && \text{(by Corollary 47)} \\
&\mapsto^{+}_{pr1} =_{ad} \mathcal{C}_a(\!|E|\!)[\mathcal{C}_a[\![N']\!](\mathcal{K}(\!|E|\!) : K)] && \text{(by Lemma 54)} \\
&\leftarrow\!\!\!\dashv^{\star}_{ad} \mathcal{C}_a[\![E[N']]\!]K && \text{(by Corollary 47)} \\
&\equiv \mathcal{C}_a[\![N]\!]K
\end{aligned}$$
□

Finally, we use Lemma 45 to generalize, completing the third step:

Lemma 56. *If $M \mapsto N$ and $M \sim P$, then there is Q such that $P \mapsto^{+} Q$ and $N \sim Q$.*

Proof. We have $M \sim P$, so $P =_{ad} \mathcal{C}_a[\![]\!]M\,\mathbf{ret}$. $M \mapsto N$, so $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} \mapsto^{+}_{pr1} =_{ad} \mathcal{C}_a[\![]\!]N\,\mathbf{ret}$ by Proposition 55. So $P =_{ad} \mapsto^{+}_{pr1} =_{ad} \mathcal{C}_a[\![]\!]N\,\mathbf{ret}$. Since $\mapsto^{+}_{pr1}$ is short for $\mapsto^{\star}_{ad} \mapsto_{pr} \mapsto^{\star}_{ad}$, we have $P =_{ad} \mapsto_{pr} =_{ad} \mathcal{C}_a[\![]\!]N\,\mathbf{ret}$. Then, by Lemma 45, we have $P \mapsto^{+} Q =_{ad} \mathcal{C}_a[\![]\!]N\,\mathbf{ret}$ for some Q. □

And now we have the correctness result for the CPS transform on annotated terms:

Lemma 57. *For any λ^{a}_{need}-term M:*

1. *$M \Downarrow$ iff $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} \Downarrow$.*
2. *$M \Uparrow$ iff $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} \Uparrow$.*
3. *$M \not\Downarrow$ iff $\mathcal{C}_a[\![M]\!]\,\mathbf{ret} \not\Downarrow$.*

Proof. By Lemmas 51 to 53 and 56, using determinacy. □

From λ_{need} to $\lambda_{cps}^{:=_1}$

From these pieces, we assemble the correctness of the call-by-need CPS transform:

Theorem 58. *For any λ_{need}-term M:*

1. *$M \Downarrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Downarrow$.*

2. *$M \Uparrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \Uparrow$.*

3. *$M \not\Downarrow$ iff $\mathcal{C}[\![M]\!]\,\mathbf{ret} \not\Downarrow$.*

Proof. Immediate from Propositions 40 and 41 and Lemma 57. □

Naming and π-encoding

Since we have introduced some of the naming mechanism into the "unnamed" CPS language $\lambda_{cps}^{:=_1}$, the simulation proof for the naming transform needs to be more subtle. The readback function $\mathcal{N}^{-1}$ we defined before undoes *all* assignments; now that the source CPS calculus is only *mostly* unnamed, this is too blunt an instrument. Instead, we will use a relation that can *selectively* eliminate sharing.

First, we need to restrict our terms so that we can reason about what variables may be assigned to.

Definition 59. *A $\lambda_{cps}^{:=_1}$ term or value is* localized *if no variable appearing on the left of an assignment subterm is bound by a λ.*[11]

For example, the value $\lambda(x,y).\, x := \lambda k.\, ky$ **in** kx is not allowed, since x is bound by the λ. Since free and ν-bound variables are not subject to

[11]We borrow the term *localized* from the π-calculus literature. The localized π-calculus is a subcalculus that forbids processes from listening on channels they have received from other processes. As in the π-calculus, this restriction can be made finer using a type system.

substitution by β-reduction, localized terms are closed under reduction. In particular, we can say with certainty which variables in a localized term may ever be assigned to, no matter the context—only those assigned to by *subterms* of the term.

Proposition 60. *If K is localized, then $\mathcal{C}_a[\![M]\!]K$ is localized. In particular, $\mathcal{C}_a[\![M]\!]\,\mathbf{ret}$ is localized.*

Proof. Easy induction on M. □

Now, keeping in mind that $\lambda^{:=_1}_{cps,vn}$ is a subset of $\lambda^{:=_1}_{cps}$:

Definition 61. *The relation $\prec$ is the restriction to $\lambda^{:=_1}_{cps} \times \lambda^{:=_1}_{cps,vn}$ of the reflexive, transitive, and congruent closure of the following rules on localized terms:*

$$M\{\lambda y^+.\, N/x\} \prec \nu x.\, x := \lambda y^+.\, N \;\mathbf{in}\; M$$

(if x not assigned in M)

$$M\{\lambda_1 y^+.\, N/x\} \prec \nu x.\, x :=_1 \lambda y^+.\, N \;\mathbf{in}\; M$$

(if x is affine and not assigned in M or N)

$$M \prec \nu x.\, M$$

(if x not free in M)

This suffices to prove the correctness of the naming transform:

Lemma 62. *For any $\lambda^{:=_1}_{cps}$-term M and variable k:*

1. *$M \Downarrow$ iff $\mathcal{N}[\![M]\!] \Downarrow$.*

2. *$M \Uparrow$ iff $\mathcal{N}[\![M]\!] \Uparrow$.*

3. *$M \not\Downarrow$ iff $\mathcal{N}[\![M]\!] \not\Downarrow$.*

Proof. As before, we show that $M \prec \mathcal{N}[\![M]\!]$ and that $\prec$ is a backward bisimulation that preserves outcomes in the backward direction. The result follows as always from these observations and determinism.

Initial Condition That $M \prec \mathcal{N}[\![M]\!]$ can be found by an easy induction, as $\prec$ simply undoes the manipulations performed by $\mathcal{N}$.

Simulation We need that $M \prec P$ and $P \mapsto Q$ imply that there is N with $M \mapsto^\star N$ and $N \prec Q$.

If $P \mapsto Q$, then we have a reduction by β, *deref*, or *deref*$_1$. β-reductions are unchanged by $\prec$. For *deref*, we have $P \equiv E[f := \lambda(x^+).\, P' \textbf{ in } E'[f(y^+)]$ and $Q \equiv E[f := \lambda(x^+)P' \textbf{ in } E'[(\lambda(x^+).\, P')(y^+)]]$. Now consider how M might relate to P: Applications of $\prec$ inside E, M, or E' would not interfere with the *deref*-reduction (we can apply the rules in Q instead). If f was substituted into the body of P, then we can simply take $M = N$. Otherwise, we can contract M to find N; either way, $N \prec Q$.

The case for *deref*$_1$ (i.e.for an ephemeral assignment rather than a permanent one) is similar, only to get $N \prec Q$ at the end, we need to apply the third rule of $\prec$ to collect the ν as garbage.

Outcomes As with Corollary 35, A and s are invariant under $\prec$, and every *deref* (or *deref*$_1$) creates a standard β-redex, so answers, stuck states, and divergence are preserved. □

The augmentation of $\mathcal{P}$ for ephemeral assignment is easy to prove correct:

Lemma 63. *For any $\lambda^{:=_1}_{cps,vn}$-term M, $M \Downarrow$ iff $\mathcal{P}[\![M]\!] \Downarrow_{\overline{\mathbf{ret}}}$*

Proof. To the proof of Lemma 36, we need only add consideration of ephemeral assignment, which corresponds just as strongly as permanent assignment. □

Finally we have our proof[12]of the correctness of the call-by-need π-encoding:

Theorem 64. *For any λ_{need}-term M, M reduces to an answer iff* $\mathcal{E}[\![M]\!]_{\mathbf{ret}} \Downarrow_{\overline{\mathbf{ret}}}$.

Proof. Immediate from Theorem 58 and Lemmas 62 and 63, since $\mathcal{E}[\![M]\!]_k \triangleq \mathcal{P}[\![\mathcal{N}[\![\mathcal{C}[\![M]\!]k]\!]]\!]$. □

Abstract Machine

Just as we did with the uniform CPS transform, we can derive an abstract machine from the call-by-need transform. First, we represent ephemeral assignment in store-passing style: A thunk assigned ephemerally should be erased from store when it is accessed. We use the symbol $\perp$ to denote such a "missing" value; the store will bind $\perp$ to a variable that has been allocated (by a ν) but currently has no value.

Using this representation, the functional correspondence gives us the abstract machine in Fig. 41. There are different machine states for examining a term, a thunk, a continuation, or a closure, and a halt state returning the final value and store. The store is a map from *locations* to thunks, and the environment maps local variables to locations in the store.

Notably, up to a few transition compressions, this abstract machine is the same as one derived by Ager, Danvy, and Midtgaard (Ager et al., 2004)[13], except that when a suspended computation is retrieved from the environment, it is removed. In this way, it resembles the original call-by-need abstract machine by Sestoft (Sestoft, 1997). Without a **letrec** form in the source, however,

[12]This is not the first such proof (Brock & Ostheimer, 1995), though previous proofs did not exploit the connection to CPS.

[13]Specifically, it resembles the first variant mentioned in section 3 of (Ager et al., 2004).

$$\begin{array}{rrl}
\text{Locations:} & & \ell, \ldots \\
\text{Terms:} & M, N ::= & x \mid \lambda x.\, M \mid MN \\
\text{Thunks:} & t ::= & KSuspM, \ell, \rho \mid KMemof \mid \bot \\
\text{Continuations:} & k ::= & KRet \mid KApplyM, \rho, k \mid KUpdate\ell, k \\
\text{Closures:} & f ::= & KClosx, M, \rho \\
\text{Stores:} & \sigma ::= & \epsilon \mid \sigma[\ell = t] \\
\text{Environments:} & \rho ::= & \epsilon \mid \rho[x = \ell] \\
\text{States:} & S ::= & \langle M, \sigma, \rho, k \rangle_M \mid \langle t, \sigma, k \rangle_T \\
& & \mid \langle k, f, \sigma \rangle_K \mid \langle f, \ell, \sigma, k \rangle_F \\
& & \mid \langle f, \sigma \rangle_H
\end{array}$$

$$M \mapsto \langle M, \epsilon, \epsilon, KRet \rangle_M$$

$$\begin{array}{rl}
\langle x, \sigma, \rho, k \rangle_M & \mapsto \langle t, \sigma, k \rangle_T \\
\text{where } \rho(x) \equiv \ell \text{ and } \sigma(\ell) \equiv t & \\
\langle \lambda x.\, M, \sigma, \rho, k \rangle_M & \mapsto \langle k, KClosx, M, \rho, \sigma \rangle_K \\
\langle MN, \sigma, \rho, k \rangle_M & \mapsto \langle M, \sigma, \rho, KApplyN, \rho, k \rangle_M \\
 & \\
\langle KSuspM, \ell, \rho, \sigma, k \rangle_T & \mapsto \langle M, \sigma[\ell = \bot], \rho, KUpdate\ell, k \rangle_M \\
\langle KMemof, \sigma, k \rangle_T & \mapsto \langle k, f, \sigma \rangle_K \\
 & \\
\langle KRet, f, \sigma \rangle_K & \mapsto \langle f, \sigma \rangle_H \\
\langle KApplyM, \rho, k, f, \sigma \rangle_K & \mapsto \langle f, \ell, \sigma[\ell = KSuspM, \ell, \rho], k \rangle_F \\
\text{where } \ell \notin \sigma & \\
\langle KUpdate\ell, k, f, \sigma \rangle_K & \mapsto \langle k, f, \sigma[\ell = KMemof] \rangle_K \\
 & \\
\langle KClosx, M, \rho, \ell, \sigma, k \rangle_F & \mapsto \langle M, \sigma, \rho[x = \ell], k \rangle_M
\end{array}$$

FIGURE 41. The abstract machine derived from $\mathcal{C}$.

this difference in behavior cannot be observed, since the symbol binding a computation cannot appear free in the term being computed.

It is also quite similar to one derived recently by Danvy and Zerny (Danvy & Zerny, 2013), which they call the lazy Krivine machine. The mechanisms are superficially different in a few ways. For them, a thunk is simply an unevaluated term, whereas we remember whether the thunk has been evaluated before (and a few bookkeeping details). However, this is merely a different choice for the division of responsibility: We hand control to the thunk, and then the thunk determines whether to set up an update or simply return a value. The lazy Krivine machine instead inspects the thunk when it is retrieved: If it is a value, it is returned immediately, and otherwise it is evaluated. Hence our thunks are *tagged* and theirs are not. The tags are largely an artifact of the connection to the π-calculus translation—whether an argument has been evaluated is evident from the structure of the process representing it. Another difference is that the lazy Krivine machine lacks an environment, relying entirely on the store, but again this is superficial.

CHAPTER V

Conclusion

As we have seen, there are many tradeoffs in designing an intermediate language. Some are clear-cut, such as how much low-level detail to expose—too little, and the optimizer cannot make important decisions about safety and efficiency; too much, and the IL becomes unworkable to implement or to reason about. Other design points are subtle, such as the use of continuations in a functional IL. Convenient approximations, such as functions as join points, may be hazardous. We hope to have found the “sweet spot” by translating from Sequent Core back to the established Core language.

The right extension to an intermediate language can enable singnificant new optimizations, and translation offers a powerful method of deriving the properties of the new extension with minimal effort while making the correctness as clear as possible. Reasoning in one language while implementing another lets us exploit the power of theory without overly conceding on matters of engineering.

In particular, we were able to make improvements to a mature compiler. Compared to the baseline of System F, F_J is a rather small change; other transformations are barely affected; the new commuting conversions are valuable in practice; and they make the transformation pipeline more robust.

Although we have presented F_J as a lazy language, *everything in this paper applies equally to a call-by-value language.* All one needs to do is to change the evaluation context, the notion of what is substitutable, and a few typing rules.

For example, this expression does not match the *casefloat* axiom:

$$f \begin{pmatrix} \textbf{case}\, g\ x\ y\, \textbf{of}\, A \to e_1 \\ \hfill B \to e_2 \end{pmatrix}$$

Nor do we want it to be: the function f is presumed lazy, so the **case** expression—and hence the call to g, which may be expensive—may never happen. If, however, we decide that $f\ \square$ *is* a valid evaluation context, then we can reduce by *casefloat* to:

$$\begin{aligned} &\textbf{case}\, g\ x\ y\, \textbf{of}\, A \to f\ e_1 \\ &\qquad\qquad\quad\;\; B \to f\ e_2 \end{aligned}$$

The work on the π-calculus in Chapter IV suggests another extension to GHC's Core language. Since Core doesn't have primitives expressing the updates that cache call-by-need computation, the simplifier and other main optimization passes don't have any opportunity to move, combine, or eliminate them. It is my hope that these operations would find synergies with other passes in much the same way the new case-of-case transformation enables list fusion to perform better.

More generally, the connection between CPS transforms, the π-calculus, and graph reduction had been considered only in the call-by-value and call-by-name worlds. We have seen that only a modest extension to the target CPS calculus is required in order to put call-by-need on an equal footing. Hopefully, we can find further uses for the constructive update calculus; for instance, given the closeness to π-calculus channel operations, it is possible that other π-calculus encodings can be reinterpreted as CPS transforms as well. In another direction, since many type systems for the π-calculus have been proposed (Sangiorgi & Walker, 2003), it seems worth exploring whether we can use the techniques outlined here to consider a typed CPS transform and a typed encoding into the π-calculus.

References Cited

Accattoli, B. (2013). Evaluating functions as processes. In *Termgraph* (pp. 41–55).

Ager, M. S., Danvy, O., & Midtgaard, J. (2004). A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, *90*(5), 223–232.

Ahmed, A. & Blume, M. (2008). Typed closure conversion preserves observational equivalence. In J. Hook & P. Thiemann (Eds.), *ICFP 2008* [Proceedings of the 13th ACM SIGPLAN international conference on functional programming], September 20–28, 2008 (pp. 157–168). Victoria, BC, Canada. ACM. doi:10.1145/1411204.1411227

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Amadio, R. (2011). *A decompilation of the pi-calculus and its application to termination.* (Tech. rep. No. UMR CNRS 7126). Université Paris Diderot.

Appel, A. W. (1992). *Compiling with Continuations*. Cambridge University Press.

Appel, A. W. (1998a). *Modern Compiler Implementation in ML*. Cambridge University Press.

Appel, A. W. (1998b). SSA is functional programming. *SIGPLAN Notices*, *33*(4), 17–20. doi:10.1145/278283.278285

Appel, C., van Oostrom, V., & Simonsen, J. G. (2010). Higher-order (non-)modularity. In *Rta* (pp. 17–32).

Ariola, Z. M. & Klop, J. W. (1996). Lambda calculus with explicit recursion. *Information and Computation*, *139*.

Ariola, Z. M. & Felleisen, M. (1997). The call-by-need lambda calculus. *J. Functional Programming*, *7*(3), 265–301.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., & Wadler, P. (1995). A call-by-need lambda calculus. In R. K. Cytron & P. Lee (Eds.), *POPL '95* [Conference record of POPL '95: 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages], January 23–25, 1995 (pp. 233–246). San Francisco, California, USA. ACM Press. doi:10.1145/199448.199507

Baker-Finch, C. A., Glynn, K., & Peyton Jones, S. L. (2004). Constructed product result analysis for Haskell. *J. Funct. Program. 14*(2), 211–245. doi:10.1017/S0956796803004751

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics. Amsterdam: North Holland.

Bove, A., Dybjer, P., & Norell, U. (2009). A brief overview of Agda: A functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, & M. Wenzel (Eds.), *TPHOLs 2009* [Theorem proving in higher order logics, 22nd international conference], August 17–20, 2009 (Vol. 5674, pp. 73–78). Lecture Notes in Computer Science. Munich, Germany. Springer. doi:10.1007/978-3-642-03359-9_6

Brady, E. (2013). Idris, a general-purpose dependently typed programming language: design and implementation. *J. Funct. Program. 23*(5), 552–593. doi:10.1017/S095679681300018X

Brock, S. & Ostheimer, G. (1995). Process semantics of graph reduction. In *Concur* (pp. 471–485).

Chakravarty, M. M. T., Keller, G., & Peyton Jones, S. (2005). Associated type synonyms. In O. Danvy & B. C. Pierce (Eds.), *ICFP 2005* [Proceedings of the 10th ACM SIGPLAN international conference on functional programming], September 26–28, 2005 (pp. 241–253). Tallinn, Estonia. ACM. doi:10.1145/1086365.1086397

Chakravarty, M. M. T., Keller, G., & Zadarnowski, P. (2003). A functional perspective on SSA optimisation algorithms. *Electr. Notes Theor. Comput. Sci. 82*(2), 347–361. doi:10.1016/S1571-0661(05)82596-4

Chang, S. & Felleisen, M. (2012). The call-by-need lambda calculus, revisited. In *Programming languages and systems* (pp. 128–147). Springer.

Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics*, *2*, 33, 346–366.

Coutts, D., Leshchinskiy, R., & Stewart, D. (2007, October). Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th acm sigplan international conference on functional programming* (pp. 315–326). ICFP '07. Freiburg, Germany: ACM. doi:10.1145/1291151.1291199

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*(4), 451–490. doi:10.1145/115372.115320

Danvy, O. & Filinski, A. (1989). *A Functional Abstraction of Typed Contexts.* (Tech. rep. No. 89/12). DIKU, University of Copenhagen, Copenhagen, Denmark.

Danvy, O. & Filinski, A. (1992). Representing control: A study of the CPS transformation. *Mathematical structures in computer science*, *2*(04), 361–391.

Danvy, O. & Nielsen, L. R. (2003). A first-order one-pass cps transformation. *Theoretical Computer Science*, *308*(1), 239–257.

Danvy, O. & Zerny, I. (2013). A synthetic operational account of call-by-need evaluation. In *Proceedings of the 15th symposium on principles and practice of declarative programming* (pp. 97–108). ACM.

Davidson, J. W. & Fraser, C. W. (1980). The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst. 2*(2), 191–202. doi:10.1145/357094.357098

Downen, P., Maurer, L., Ariola, Z. M., & Peyton Jones, S. (2016). Sequent calculus as a compiler intermediate language. In *Icfp 2016* [Proceedings of the 21st ACM SIGPLAN international conference on functional programming], September 18–22, 2016 (pp. 74–88). Nara, Japan. doi:10.1145/2951913.2951931

Downen, P., Maurer, L., Ariola, Z. M., & Varacca, D. (2014). Continuations, processes, and sharing. In O. Chitil, A. King, & O. Danvy (Eds.), *Proceedings of the 16th international symposium on principles and practice of declarative programming, kent, canterbury, united kingdom, september 8-10, 2014* (pp. 69–80). ACM. doi:10.1145/2643135.2643155

Felleisen, M. & Friedman, D. (1986). Control operators, the SECD-machine, and the λ-calculus. In *Formal descriptions of programming concepts* (pp. 193–219).

Flanagan, C., Sabry, A., Duba, B. F., & Felleisen, M. (1993). The essence of compiling with continuations. In R. Cartwright (Ed.), *PLDI 1993* [Proceedings of the ACM SIGPLAN '93 conference on programming language design and implementation], June 23–25, 1993 (pp. 237–247). Albuquerque, New Mexico, USA. ACM. doi:10.1145/155090.155113

Fluet, M. & Weeks, S. (2001). Contification using dominators. In B. C. Pierce (Ed.), *ICFP '01* [Proceedings of the sixth ACM SIGPLAN international conference on functional programming], September 3–5, 2001 (pp. 2–13). Firenze (Florence), Italy. ACM. doi:10.1145/507635.507639

Gentzen, G. (1935). Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, *39*(1), 176–210. doi:10.1007/BF01201353

Gill, A. & Hutton, G. (2009). The worker/wrapper transformation. *J. Funct. Program. 19*(2), 227–251. doi:10.1017/S0956796809007175

Girard, J.-Y., Taylor, P., & Lafont, Y. (1989). *Proofs and types.* Cambridge University Press Cambridge.

Hatcliff, J. & Danvy, O. (1997). Thunks and the λ-calculus. *J. Funct. Program. 7*(3), 303–319.

Herbelin, H. (1995). A λ-calculus structure isomorphic to gentzen-style sequent calculus structure. In L. Pacholski & J. Tiuryn (Eds.), *CSL '94* [Computer science logic, 8th international workshop, selected papers], September 25–30, 1994 (Vol. 933, pp. 61–75). Lecture Notes in Computer Science. Kazimierz, Poland. Springer. doi:10.1007/BFb0022247

Hinze, R. (2005). Church numerals, twice! *J. Funct. Program. 15*(1), 1–13. doi:10.1017/S0956796804005313

Howard, W. A. (1980). The formulae-as-types notion of construction. In J. Seldin & J. Hindley (Eds.), *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism.* New York: Academic Press.

Keep, A., Hearn, A., & Dybvig, R. (2021). Optimizing closures in O(0) time. In *Annual workshop on scheme and functional programming.* ACM.

Kelsey, R. (1995). A correspondence between continuation passing style and static single assignment form. In M. D. Ernst (Ed.), *IR'95* [Proceedings of the ACM SIGPLAN workshop on intermediate representations], January 22, 1995 (pp. 13–23). San Francisco, CA, USA. ACM. doi:10.1145/202529.202532

Kennedy, A. (2007). Compiling with continuations, continued. In R. Hinze & N. Ramsey (Eds.), *ICFP 2007* [Proceedings of the 12th ACM SIGPLAN international conference on functional programming], October 1–3, 2007 (pp. 177–190). Freiburg, Germany. ACM. doi:10.1145/1291151.1291179

Krivine, J.-L. (2007). A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, *20*(3), 199–207.

Lattner, C. & Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO 2004* [Proceedings of the

2nd IEEE / ACM international symposium on code generation and optimization], March 20–24, 2004 (pp. 75–88). San Jose, CA, USA. IEEE Computer Society. doi:10.1109/CGO.2004.1281665

Leroy, X. (2009). A formally verified compiler back-end. *Journal of Automated Reasoning*, *43*(4), 363–446.

Lindley, S. (2005). *Normalisation by evaluation in the compilation of typed functional programming languages* (Doctoral dissertation, University of Edinburgh, College of Science and Engineering, School of Informatics).

Lowry, E. S. & Medlock, C. W. (1969). Object code optimization. *Commun. ACM*, *12*(1), 13–22. doi:10.1145/362835.362838

Marlow, S., Yakushev, A. R., & Peyton Jones, S. (2007). Faster laziness using dynamic pointer tagging. In R. Hinze & N. Ramsey (Eds.), *ICFP 2007* [Proceedings of the 12th ACM SIGPLAN international conference on functional programming], October 1–3, 2007 (pp. 277–288). Freiburg, Germany. ACM. doi:10.1145/1291151.1291194

Maurer, L., Downen, P., Ariola, Z. M., & Peyton Jones, S. (2017). Compiling without continuations. In A. Cohen & M. T. Vechev (Eds.), *PLDI 2017* [Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation], June 18–23, 2017 (pp. 482–494). Barcelona, Spain. ACM. doi:10.1145/3062341.3062380

Meyer, A. & Cosmadakis, S. (1988, July). *Semantical Paradigms: Notes for an Invited Lecture*. MIT Laboratory for Computer Science. 545 Technology Square, Cambridge, MA 02139.

Milner, R. (1992). Functions as processes. *Mathematical Structures in Computer Science*, *2*(02), 119–141.

Milner, R., Parrow, J., & Walker, D. (1992). A calculus of mobile processes, I. *Information and Computation*, *100*(1), 1–40. doi:10.1016/0890-5401(92)90008-4

Milner, R. & Sangiorgi, D. (1992). Barbed bisimulation. In *Automata, languages and programming* (pp. 685–695). Springer.

Minamide, Y., Morrisett, J. G., & Harper, R. (1996). Typed closure conversion. In H. Boehm & G. L. S. Jr. (Eds.), *POPL '96* [Conference record of POPL '96: the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages, papers presented at the symposium], January 21–24, 1996 (pp. 271–283). St. Petersburg Beach, Florida, USA. ACM Press. doi:10.1145/237721.237791

Morrisett, J. G., Walker, D., Crary, K., & Glew, N. (1999). From system F to typed assembly language. *ACM Trans. Program. Lang. Syst. 21*(3), 527–568. doi:10.1145/319301.319345

Novillo, D. (2003). Tree SSA: A new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC developers' summit* (pp. 181–193).

Okasaki, C., Lee, P., & Tarditi, D. (1994). Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, *7*(1), 57–82.

LLVM. (2015). LLVM language reference manual.

Parigot, M. (1992). $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In A. Voronkov (Ed.), *Logic programming and automated reasoning* (Vol. 624, pp. 190–201). Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/BFb0013061

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages.* Prentice-Hall.

Peyton Jones, S. L. & Launchbury, J. (1991). Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on functional programming languages and computer architecture* (pp. 636–666). London, UK, UK: Springer-Verlag.

Peyton Jones, S. L. & Santos, A. L. M. (1998). A transformation-based optimiser for Haskell. *Sci. Comput. Program. 32*(1-3), 3–47. doi:10.1016/S0167-6423(97)00029-4

Peyton Jones, S. & Meijer, E. (1997). Henk: A typed intermediate language. In *TIC 1997*, June 8, 1997. Amsterdam, The Netherlands.

Peyton Jones, S., Vytiniotis, D., Weirich, S., & Washburn, G. (2006). Simple unification-based type inference for gadts. In J. H. Reppy & J. L. Lawall (Eds.), *ICFP 2006* [Proceedings of the 11th ACM SIGPLAN international conference on functional programming], September 16–21, 2006 (pp. 50–61). Portland, Oregon, USA. ACM. doi:10.1145/1159803.1159811

Peyton Jones, S., Partain, W., & Santos, A. (1996, May). Let-floating: Moving bindings to give faster programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96).* Philadelphia: ACM.

Peyton Jones, S. & Santos, A. (1998, September). A transformation-based optimiser for Haskell. *Science of Computer Programming*, *32*(1-3), 3–47.

Pierce, B. C. (2002). *Types and programming languages.* MIT Press.

Plotkin, G. D. (1975). Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci. 1*(2), 125–159. doi:10.1016/0304-3975(75)90017-1

Plotkin, G. D. (1977). LCF considered as a programming language. *Theor. Comput. Sci. 5*(3), 223–255. doi:10.1016/0304-3975(77)90044-5

Pop, S. (2006). *The SSA representation framework: semantics, analyses and GCC implementation* (Doctoral dissertation, École Nationale Supérieure des Mines de Paris).

Prosser, R. T. (1959). Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1–3, 1959, Eastern joint IRE-AIEE-ACM computer conference* (pp. 133–138). ACM.

Reppy, J. H. (2002). Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation*, *15*(2-3), 161–180. doi:10.1023/A:1020839128338

Reynolds, J. C. (1993). The discoveries of continuations. *Lisp and Symbolic Computation*, *6*(3-4), 233–248.

Reynolds, J. C. (1998). Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, *11*(4), 363–397. doi:10.1023/A:1010027404223

Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1988). Global value numbers and redundant computations. In J. Ferrante & P. Mager (Eds.), *POPL '88* [Proceedings of the fifteenth annual ACM SIGPLAN-SIGACT symposium on principles of programming languages], January 10–13, 1988 (pp. 12–27). San Diego, California, USA. ACM Press. doi:10.1145/73560.73562

Sabry, A. & Felleisen, M. (1993). Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, *6*(3-4), 289–360.

Sabry, A. & Wadler, P. (1997). A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *19*(6), 916–941.

Sangiorgi, D. (1999, July). From λ to π; or, rediscovering continuations. *Mathematical Structures in Computer Science*, *9*(4), 367–401.

Sangiorgi, D. & Walker, D. (2003). *The π-calculus: a Theory of Mobile Processes*. Cambridge Univ. Press.

Santos, A. L. M. (1995). *Compilation by Transformation in Non-Strict Functional Languages* (Doctoral dissertation, University of Glasgow).

Sestoft, P. (1997). Deriving a lazy abstract machine. *Journal of Functional Programming*, *7*(03), 231–264.

Steele, Jr., G. L. & Sussman, G. J. (1976, March). *Lambda: The Ultimate Imperative.* (AI Memo No. AIM-353). Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Steele, G. L., Jr. (1978, May). *Rabbit: A Compiler for Scheme.* (Technical Report No. AI-TR-474). Artificial Intelligence Laboratory, MIT. Cambridge, MA.

Sulzmann, M., Chakravarty, M. M. T., Jones, S. L. P., & Donnelly, K. (2007). System F with type equality coercions. In F. Pottier & G. C. Necula (Eds.), *TLDI '07* [Proceedings of the 2007 ACM SIGPLAN international workshop on types in language design and implementation], January 16, 2007 (pp. 53–66). Nice, France. ACM. doi:10.1145/1190315.1190324

Sussman, G. J. & Steele, Jr., G. L. (1975, December). *SCHEME: An interpreter for untyped lambda-calculus.* (AI Memo No. AIM-349). Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Svenningsson, J. (2002, September). Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP '02* [Proceedings of the seventh ACM SIGPLAN international conference on functional programming], October 4–6, 2002 (pp. 124–132). Pittsburgh, Pennsylvania, USA. Pittsburgh: ACM. doi:10.1145/581478.581491

Tolmach, A. P. & Oliva, D. (1998). From ML to Ada: Strongly-typed language interoperability via source translation. *J. Funct. Program. 8*(4), 367–412.

Xi, H., Chen, C., & Chen, G. (2003). Guarded recursive datatype constructors. In A. Aiken & G. Morrisett (Eds.), *POPL 2003* [Conference record of POPL 2003: the 30th SIGPLAN-SIGACT symposium on principles of programming languages], January 15–17, 2003 (pp. 224–235). New Orleans, Louisiana, USA. ACM. doi:10.1145/640128.604150

Xi, H. & Pfenning, F. (1999). Dependent types in practical programming. In A. W. Appel & A. Aiken (Eds.), *POPL '99* [Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages], January 20–22, 1999 (pp. 214–227). San Antonio, TX, USA. ACM. doi:10.1145/292540.292560