Intermediate Languages for Optimization

Luke Maurer

December 12, 2015

1 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.

The subtlest purpose of an IR, however, is to empower optimization. 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) and *functional* languages (Section 3) separately. In Section 4, we will look at the specific challenges facing GHC, stemming from the particular features of the lazy functional language Haskell. Then, in Section 5, we introduce our own IL, currently being implemented in a patch to GHC. We conclude in Section 6.

2 Languages for Imperative Optimizations

2.1 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* [DF80]. 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.

2.2 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 languages because the static single-assignment form (Section 2.3) 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.

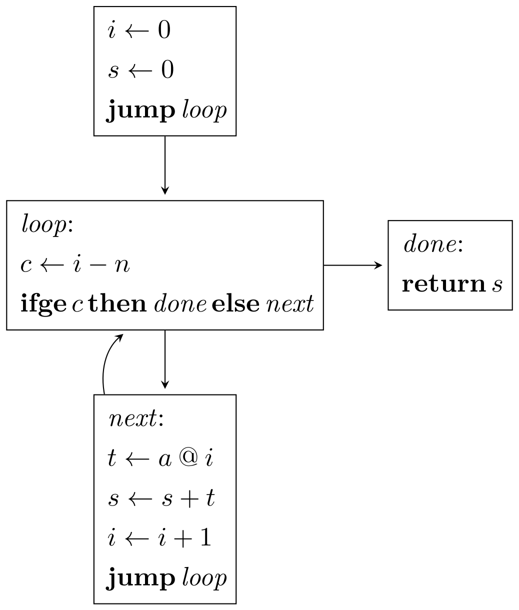


Figure 1: The control-flow graph for a simple array-sum program.

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

Figure 2: An example of common-subexpression elimination.

2.3 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 [App98a; ASU86].

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 the above example, $a + 3$ is not a 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':

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

We have 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.

```
  $i_0 \leftarrow 0$
  $s_0 \leftarrow 0$
$loop$:
  $i \leftarrow \phi(i_0,\ i')$
  $s \leftarrow \phi(s_0,\ s')$
  $c \leftarrow i - n$
  **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.

2.3.1 The ϕ-Node

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

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

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* [Cyt+91] 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 [RWZ88] was a generalization of CSE called *global value numbering*: once we *can* trust the face value of an expression

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

Figure 4: The syntax of the untyped λ-calculus.

$a + b$ 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 [Nov03; Pop06], and the LLVM framework's bytecode language is in SSA form [LA04; LLVM15].

3 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.

3.1 The λ-Calculus

Three-address codes represent imperative programs by a minimum of constructs. Similarly, Church's *λ-calculus* [Bar84] 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 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 [Hin05], 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 [Ari+95]. For **let rec**, in particular, one has to be careful; see, for instance, Pierce [Pie02, 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, 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:

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

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:

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

[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.

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*:

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

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

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

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

Languages such as ML and Haskell also allow for more complex patterns, but those can be re-expressed using these simple ones [Pey87]. 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 [Plo75; Ari+95], 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:

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

[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.

Here V stands for a *value*, which must be a λ-abstraction or a constant. This 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 [Ari+95], 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 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 [App92, 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 [PL91; San95; PS98].

3.2 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.

[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**.

Continuation-passing style [SS75], or CPS, is a way for the λ-calculus to play much the same role for functional programs. As proved formally by Plotkin [Plo75], a term written in CPS effectively specifies its own evaluation order, and any λ-term can be translated into CPS 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* [App92, 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.

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

Figure 6: A call-by-value CPS transform.

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

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

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:

$$
\begin{array}{lcl}
\mathit{sum}\ \mathit{xs} & = & \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
& & \quad \mathit{Nil} \quad\quad\;\; \to\ 0 \\
& & \quad \mathit{Cons}\ x\ \mathit{xs}' \to\ x\ +\ \mathit{sum}\ \mathit{xs}'
\end{array}
$$

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

Its CPS form (after some simplification) is:

$$
\begin{array}{lll}
sum \;=\; \lambda xs.\, \lambda k.\, \textbf{case}\; xs\; \textbf{of} & & \\
\qquad\qquad\qquad Nil & \rightarrow & k\, 0 \\
\qquad\qquad\qquad Cons\; x\; xs' & \rightarrow & sum\; xs'\; (\lambda y.\, k\; (x\; +\; y))
\end{array}
$$

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 $x\; +\; sum\; xs$, the first computation that would be made is the recursive call $sum\; xs$, 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.)

3.3 Administrative Normal Form

CPS is expressive but heavyweight. A less radical alternative with many, though not all [Ken07], 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.

ANF emerges naturally by considering the *inverse* of the CPS transform [Fla+93]. 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:

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

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, SSA conversion (see Section 3.4.2) works broadly the same way [CKZ03].

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

Figure 8: The ANF language produced by the CPS language in Fig. 7 by an inverse CPS transform.

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 5 when we introduce the sequent calculus.

3.4 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.

3.4.1 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

```
  $i \leftarrow i_0$
  $s \leftarrow 0$
$loop$:
  $h \leftarrow i_0 + k$
  $c \leftarrow i - h$
  **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.

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.

3.4.2 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 [Kel95; App98b].

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 [Rey98]. 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 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.

Nesting: The Dominance Tree 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.

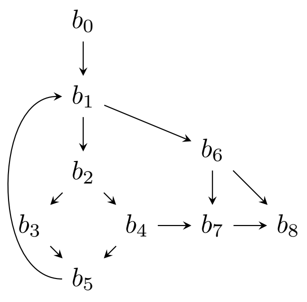
SSA programs obey a similar invariant: each variable's definition *dominates* each of its uses [App98a]. A block b_1 dominates a block b_2 if each path in the CFG from the start of the program to b_2 goes through b_1 [Pro59]. If we make each block a function, then, we should be able to satisfy the scoping 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 5.1.

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 [Cyt+91].

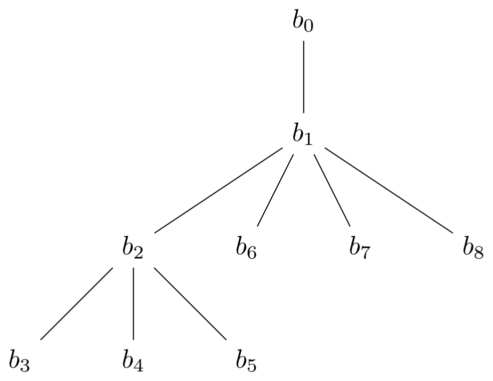
Figure 10 demonstrates the dominance tree for a somewhat tangled section of code (Fig. 10a). The CFG is shown in Fig. 10b and the dominance tree in Fig. 10c. Notice the impact of the edge from b_4 to b_7. Since a dominator must

```
$b_0$:
  $i_0 \leftarrow 0$
  $s_0 \leftarrow 0$
$b_1$:
  $i \leftarrow \phi(i_0, i_1)$
  $s \leftarrow \phi(s_0, s_3)$
  $c_1 \leftarrow i - n$
  **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.



(b) The CFG for (a).



(c) The dominance tree for (a).

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

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.

The ϕ-Node as Inside-Out Function Call 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.

4 GHC

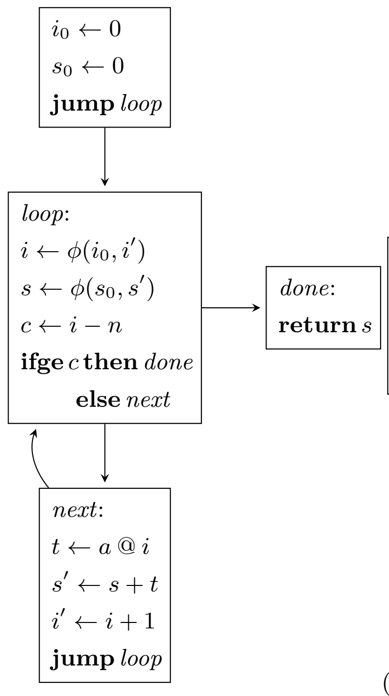
4.1 Lazy Evaluation

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 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.

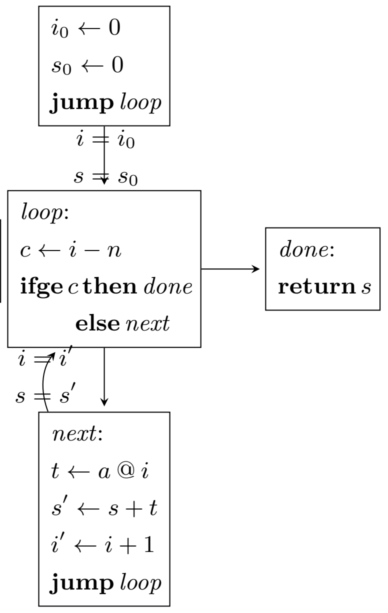
The classical way to implement lazy evaluation is to use a *memo-thunk* [HD97; OLT94; SS76]. This is a nullary function closure that will update itself when it finishes executing; on subsequent invocations, the new version will immediately return the cached answer. It is an effective strategy, and one 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

[6]Oddly, this has been proposed before as a source language construct! A 1969 paper on optimization [LM69] 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, in an era where `goto` was popular, this would have been less clear.)



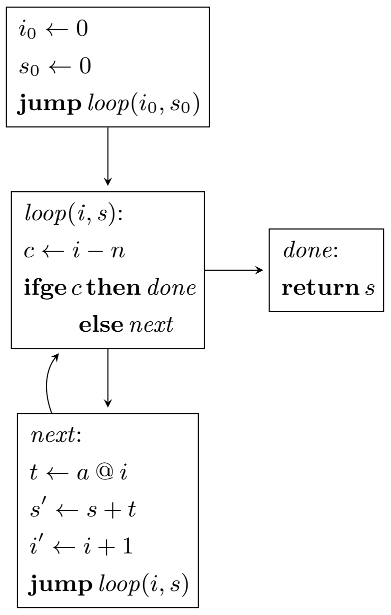
```
$i_0 \leftarrow 0$
$s_0 \leftarrow 0$
jump loop

loop:
$i \leftarrow \phi(i_0, i')$
$s \leftarrow \phi(s_0, s')$
$c \leftarrow i - n$
ifge $c$ then done
    else next

done:
return $s$

next:
$t \leftarrow a \mathbin{@} i$
$s' \leftarrow s + t$
$i' \leftarrow i + 1$
jump loop
```

(a) The CFG for Fig. 3.



```
$i_0 \leftarrow 0$
$s_0 \leftarrow 0$
jump loop

$i = i_0$
$s = s_0$

loop:
$c \leftarrow i - n$
ifge $c$ then done
    else next

done:
return $s$

$i = i'$
$s = s'$

next:
$t \leftarrow a \mathbin{@} i$
$s' \leftarrow s + t$
$i' \leftarrow i + 1$
jump loop
```

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



```
$i_0 \leftarrow 0$
$s_0 \leftarrow 0$
jump loop$(i_0, s_0)$

loop$(i, s)$:
$c \leftarrow i - n$
ifge $c$ then done
    else next

done:
return $s$

next:
$t \leftarrow a \mathbin{@} i$
$s' \leftarrow s + t$
$i' \leftarrow i + 1$
jump loop$(i, s)$
```

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

Figure 11: Moving ϕ-node values into the predecessors.

```
λa. λn.
  let rec
    b_0 = let
              i_0 = 0
              s_0 = 0
          in
          let rec
            b_1 i s = let
                          c_1 = i − n
                      in
                      let rec
                        b_2     = let
                                      c_2 = i % 2
                                  in
                                  let rec
                                    b_3     = let
                                                  t_1 = a @ i
                                                  t_2 = t_1 ∗ 2
                                                  s_1 = s + t_2
                                              in
                                              b_5 s_1
                                    b_4     = let
                                                  t_3 = a @ i
                                                  s_2 = s + t_3
                                              in
                                              if t_3 = 0 then b_7 else b_5 s_2
                                    b_5 s_3 = let
                                                  i_1 = i + 1
                                              in
                                              b_1 i_1 s_3
                                  in
                                  if c_2 = 0 then b_3 else b_4
                        b_6     = let
                                      c_3 = s − 100
                                  in
                                  if c_3 > 0 then b_7 else b_8 s
                        b_7     = let
                                      s_4 = −1
                                  in
                                  b_8 s_4
                        b_8 s_5 = s_5
                      in
                      if c_1 ≥ 0 then b_2 else b_6
          in
          b_1 i_0 s_0
  in
  b_0
```

Figure 12: The program from Fig. 10a, in ANF.

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 [MYP07]. 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.

4.2 The Core Language

There were two important design principles [PS98] 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 [PM97]: "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* [Plo77], 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 [MMH96] has been proved fully abstract [AB08], meaning not only that the conversion preserves meaning, but also that code written in the *target* language cannot "cheat" by inspecting a function's closure (perhaps to find a password). By employing a typed assembly language [Mor+98], 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.

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

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

4.3 Syntax

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

4.4 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 $\mathbf{case}\, x + y \,\mathbf{of}\, \{z \to f\, z\}$.

[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.

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

Figure 14: Types in the GHC Core language.

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.

4.5 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 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.

4.6 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 [PL91]. 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 $Int_\#$ 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 $Int_\#$ 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]

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

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:

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

[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*.

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

Next, the case-of-case transform applies twice, moving the other case all the way in:

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

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 $a_{\#}$:

$$\begin{array}{l}
\Rightarrow \textbf{case}\ x\ \textbf{of} \\
\qquad I_{\#}\ x_{\#} \rightarrow\ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad I_{\#}\ y_{\#} \rightarrow\ \textbf{case}\ x_{\#} +_{\#} y_{\#}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad s_{\#} \rightarrow\ \textbf{case}\ z\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad I_{\#}\ z_{\#} \rightarrow\ \textbf{case}\ s_{\#} +_{\#} z_{\#}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad t_{\#} \rightarrow I_{\#}\ t_{\#}
\end{array}$$

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.

4.7 Coercions and System $\mathbf{F}_C$

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 [XP99; BDN09; Bra13], 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 [Pey+06; XCC03] and type families [CKP05]. These have been accomodated in Core by the addition of a much simpler extension, the *coercion* [Sul+07]. 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.

4.7.1 Example: Well-Typed Expressions

For example, consider generalized algebric datatypes (GADTs), a popular feature 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:

```
data Expr a where
```

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

$$\begin{array}{lllll}
\mathit{Lit} & :: a & & & \to \mathit{Expr}\ a \\
\mathit{Plus} & :: \mathit{Expr}\ \mathit{Int} & \to \mathit{Expr}\ \mathit{Int} \to a \sim \mathit{Int} & & \to \mathit{Expr}\ a \\
\mathit{Eq} & :: \mathit{Expr}\ \mathit{Int} & \to \mathit{Expr}\ \mathit{Int} \to a \sim \mathit{Bool} & & \to \mathit{Expr}\ a \\
\mathit{If} & :: \mathit{Expr}\ \mathit{Bool} & \to \mathit{Expr}\ a & \to \mathit{Expr}\ a & \to \mathit{Expr}\ a
\end{array}$$

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]:

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

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 novelties are the coercions, and these are not hard.

5 Sequent Calculus for GHC

As mentioned in Section 3.3, 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 [Fla+93]. 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.

5.1 Case Floating and Join Points

When dealing with plain λ-terms, one important operation is called *case floating* [San95; PS98]. 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. An easy

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

case is when a **case** returns a function that will be applied:

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

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

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?

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

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:

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

In general, this is valid only in a lazy language; if this were a call-by-value language, we would have changed the order of evaluation by evaluating $\langle \mathrm{BIG}\rangle$ first. If b were an expression with side effects, it would take some care to rearrange the term so that **BIG** is computed at the right time and yet we can still eliminate our redex.

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:

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

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:

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

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 ⟨BIG⟩ all over again. But we can instead give ⟨BIG⟩ its own binding and (since f was chosen fresh) float it out:

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

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 [Plo75].

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 = λf. let
            arg = ⟨BIG⟩
          in
          f arg
in
case b of
  True  → j (λy. x + y)
  False → j (λy. 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.

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 = λx. let
            j = λf. let
                      arg = ⟨BIG⟩
                    in
                    f arg
          in
          case b of
            True  → j (λy. x + y)
            False → j (λy. x ∗ y)
in
case g 1 of
  ⟨HUGE⟩
```

Now suppose that this is the only reference to g (it does not appear in ⟨HUGE⟩).

[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. Code generation can either treat continuations specially and implement them using the usual call stack, or simply let them be values like any other, using heap-allocated "stack frames" and forgoing the stack entirely. By doing the latter, SML/NJ easily implements the `call/cc` operator, which allows user code to access its continuation, with little overhead [App98a, §5.9].

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:

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

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

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

Then, as before, we need to perform case floating. This time we're doing the *case-of-case* transformation [PS98; San95]. 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 ⟨BIG⟩ won't help.)

$$\begin{array}{l}
\textbf{let} \\
\quad j \ \ =\ \lambda f.\,\textbf{let} \\
\qquad\qquad\quad \mathit{arg} \ =\ \langle \text{BIG} \rangle \\
\qquad\qquad \textbf{in} \\
\qquad\qquad f\ \mathit{arg} \\
\quad j_2 =\ \lambda z.\,\textbf{case}\ z\ \textbf{of} \\
\qquad\qquad\quad \langle \text{HUGE} \rangle \\
\textbf{in} \\
\textbf{case}\ b\ \textbf{of} \\
\quad \mathit{True} \to\ j_2\ (j\ (\lambda y.\ x + y)) \\
\quad \mathit{False} \to\ j_2\ (j\ (\lambda y.\ x * y))
\end{array}$$

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.

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

Figure 15: The syntax for Sequent Core.

5.2 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, if one is not careful, it may not be simple to translate between a CPS language and Core. As when SSA was introduced to GCC [Nov03], we propose to integrate a new IL into a mature and complex compiler, so we would like to be able to interoperate with existing components as much as possible.

The *sequent calculus* was invented by Gerhard Gentzen in his study of logic, in order to prove properties of his other system, *natural deduction* [Gen35]. Decades later, it was realized that natural deduction is intimately related to the λ-calculus by what is now called the Curry–Howard isomorphism [How80]. More recently, there has been interest in the similar way that the sequent calculus can be seen as a programming language [Her95]. 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 [Par92] 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* [Ken07; FW01] 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:

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

Here it is as a term in Sequent Core:

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

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:

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

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

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

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

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

Our simplified Sequent Core term becomes:

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

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 $\langle$HUGE$\rangle$, but then it would still "ruin" j. But here, a simple μ_β-reduction, subsituting for k afterward, gives:

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

We pull the **case** into j, where it has a chance to interact with $\langle$BIG$\rangle$, 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 $\langle$BIG$\rangle$ to that of the branches $\langle$HUGE$\rangle$, 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.

5.3 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:

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

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, and again, we would like to interoperate with the large body of code extant in GHC.

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:

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

The solution is simple, as suggested by Kennedy [Ken07]. 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:

[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."

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

Thus, we can easily return to Core after working in Sequent Core. There is some overhead in translating back and forth, but early experience suggests it is tolerable.

6 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" in Sequent Core, which expresses control flow, but with a minimum of syntactic weight, and which retains the strong typing that continues to prove useful in reasoning and in compiler development.

Our preliminary implementation of Sequent Core, in the form of a new simplifier module for GHC and a few other reimplemented optimization passes, does in fact find cases much like our iterated case-lifting examples. Further avenues of inquiry include:

- What more use can we make of the knowledge exposed by the Sequent Core syntax, namely which definitions are join points versus normal functions? For instance, there is indication that *lambda-lifting*, whereby local functions are rewritten as global ones, is beneficial for many regular functions but not for continuations. We hope to implement a Sequent-Core-based lambda-lifting pass to explore this question.

- Currently, the only contification performed is similar to what GHC does in translating from core to the lower-level IL, STG. Kennedy [Ken07] and Fluet and Weeks [FW01] outline more aggressive contification algorithms. Calls to continuations are always cheaper, so we expect that the more we can contify, the better.

- As yet, we have only considered reworking various Core-to-Core optimization passes in Sequent Core, but in some ways, STG is similar to Sequent Core (it includes a *let-no-escape* declaration meant for join points). It is possible that translating straight from Sequent Core to STG would improve generated code by losing less information in translating through Core; it is very likely to make for a simpler and faster translation routine, as there would be no need for on-the-fly contification, as is currently performed.

References

[AB08] Amal Ahmed and Matthias Blume. "Typed closure conversion preserves observational equivalence". In: *ICFP 2008*. Proceedings of the 13th ACM SIGPLAN international conference on Functional programming. (Victoria, BC, Canada, Sept. 20–28, 2008). Ed. by James Hook and Peter Thiemann. ACM, 2008, pp. 157–168. ISBN: 978-1-59593-919-7.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN: 0-201-10088-6.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN: 0-521-41695-7.

[App98a] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN: 0-521-58274-1.

[App98b] Andrew W. Appel. "SSA is Functional Programming". In: *SIGPLAN Notices* 33.4 (1998), pp. 17–20.

[Ari+95] Zena M. Ariola et al. "The Call-by-Need Lambda Calculus". In: *POPL'95*. Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (San Francisco, California, USA, Jan. 23–25, 1995). Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 233–246. ISBN: 0-89791-692-1. URL: `http://dl.acm.org/citation.cfm?id=199448`.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised Edition. Studies in Logic and the Foundations of Mathematics 103. Amsterdam: North Holland, 1984. ISBN: 0-444-07508-5.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda - A Functional Language with Dependent Types". In: *TPHOLs 2009*. Theorem Proving in Higher Order Logics, 22nd International Conference. (Munich, Germany, Aug. 17–20, 2009). Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 73–78. ISBN: 978-3-642-03358-2.

[Bra13] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *J. Funct. Program.* 23.5 (2013), pp. 552–593.

[CKP05] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. "Associated type synonyms". In: *ICFP 2005*. Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming. (Tallinn, Estonia, Sept. 26–28, 2005). Ed. by Olivier Danvy and Benjamin C. Pierce. ACM, 2005, pp. 241–253. ISBN: 1-59593-064-7.

[CKZ03] Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. "A Functional Perspective on SSA Optimisation Algorithms". In: *Electr. Notes Theor. Comput. Sci.* 82.2 (2003), pp. 347–361.

[Cyt+91] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490.

[DF80] Jack W. Davidson and Christopher W. Fraser. "The Design and Application of a Retargetable Peephole Optimizer". In: *ACM Trans. Program. Lang. Syst.* 2.2 (1980), pp. 191–202.

[Fla+93] Cormac Flanagan et al. "The Essence of Compiling with Continuations". In: *PLDI 1993.* Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation. (Albuquerque, New Mexico, USA, June 23–25, 1993). Ed. by Robert Cartwright. ACM, 1993, pp. 237–247. ISBN: 0-89791-598-4.

[FW01] Matthew Fluet and Stephen Weeks. "Contification Using Dominators". In: *ICFP '01.* Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming. (Firenze (Florence), Italy, Sept. 3–5, 2001). Ed. by Benjamin C. Pierce. ACM, 2001, pp. 2–13. ISBN: 1-58113-415-0.

[Gen35] Gerhard Gentzen. "Untersuchungen über das logische Schließen. I". German. In: *Mathematische Zeitschrift* 39.1 (1935), pp. 176–210. ISSN: 0025-5874.

[HD97] John Hatcliff and Olivier Danvy. "Thunks and the λ-Calculus". In: *J. Funct. Program.* 7.3 (1997), pp. 303–319. URL: `http://journals.cambridge.org/action/displayAbstract?aid=44093`.

[Her95] Hugo Herbelin. "A λ-Calculus Structure Isomorphic to Gentzen-Style Sequent Calculus Structure". In: *CSL '94.* Computer Science Logic, 8th International Workshop, Selected Papers. (Kazimierz, Poland, Sept. 25–30, 1994). Ed. by Leszek Pacholski and Jerzy Tiuryn. Vol. 933. Lecture Notes in Computer Science. Springer, 1995, pp. 61–75. ISBN: 3-540-60017-5.

[Hin05] Ralf Hinze. "Church numerals, twice!" In: *J. Funct. Program.* 15.1 (2005), pp. 1–13.

[How80] William A. Howard. "The Formulae-as-Types Notion of Construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Ed. by J.P. Seldin and J.R. Hindley. New York: Academic Press, 1980.

[Kel95] Richard Kelsey. "A Correspondence between Continuation Passing Style and Static Single Assignment Form". In: *IR'95.* Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations. (San Francisco, CA, USA, Jan. 22, 1995). Ed. by Michael D. Ernst. ACM, 1995, pp. 13–23. ISBN: 0-89791-754-5.

[Ken07] Andrew Kennedy. "Compiling with continuations, continued". In: *ICFP 2007*. Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming. (Freiburg, Germany, Oct. 1–3, 2007). Ed. by Ralf Hinze and Norman Ramsey. ACM, 2007, pp. 177–190. ISBN: 978-1-59593-815-2.

[LA04] Chris Lattner and Vikram S. Adve. "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. (San Jose, CA, USA, Mar. 20–24, 2004). IEEE Computer Society, 2004, pp. 75–88. ISBN: 0-7695-2102-9.

[LLVM15] *LLVM language reference manual*. 2015. URL: `http://llvm.org/docs/LangRef.html`.

[LM69] Edward S. Lowry and C. W. Medlock. "Object code optimization". In: *Commun. ACM* 12.1 (1969), pp. 13–22.

[MYP07] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. "Faster laziness using dynamic pointer tagging". In: *ICFP 2007*. Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming. (Freiburg, Germany, Oct. 1–3, 2007). Ed. by Ralf Hinze and Norman Ramsey. ACM, 2007, pp. 277–288. ISBN: 978-1-59593-815-2.

[MMH96] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. "Typed Closure Conversion". In: *POPL'96*. Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium. (St. Petersburg Beach, Florida, USA, Jan. 21–24, 1996). Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. ACM Press, 1996, pp. 271–283. ISBN: 0-89791-769-3.

[Mor+98] J. Gregory Morrisett et al. "From System F to Typed Assembly Language". In: *POPL '98*. Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (San Diego, CA, USA, Jan. 19–21, 1998). Ed. by David B. MacQueen and Luca Cardelli. ACM, 1998, pp. 85–97. ISBN: 0-89791-979-3.

[Nov03] Diego Novillo. "Tree SSA A New Optimization Infrastructure for GCC". In: *Proceedings of the 2003 GCC Developers' Summit*. 2003, pp. 181–193.

[OLT94] Chris Okasaki, Peter Lee, and David Tarditi. "Call-by-Need and Continuation-Passing Style". In: *Lisp and Symbolic Computation* 7.1 (1994), pp. 57–82.

[Par92] Michel Parigot. “ -Calculus: An algorithmic interpretation of classical natural deduction”. In: *Logic Programming and Automated Reasoning*. Ed. by Andrei Voronkov. Vol. 624. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 190–201. ISBN: 978-3-540-55727-2.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[PL91] Simon L. Peyton Jones and John Launchbury. “Unboxed Values as First Class Citizens in a Non-Strict Functional Language”. In: *FPCA 1991*. Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture. (Cambridge, MA, USA, Aug. 26–30, 1991). Ed. by John Hughes. Vol. 523. Lecture Notes in Computer Science. Springer, 1991, pp. 636–666. ISBN: 3-540-54396-1.

[PS98] Simon L. Peyton Jones and André L. M. Santos. “A Transformation-Based Optimiser for Haskell”. In: *Sci. Comput. Program.* 32.1-3 (1998), pp. 3–47.

[PM97] Simon Peyton Jones and Erik Meijer. “Henk: a typed intermediate language”. In: *TIC 1997*. (Amsterdam, The Netherlands, June 8, 1997). 1997.

[Pey+06] Simon Peyton Jones et al. “Simple unification-based type inference for GADTs”. In: *ICFP 2006*. Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming. (Portland, Oregon, USA, Sept. 16–21, 2006). Ed. by John H. Reppy and Julia L. Lawall. ACM, 2006, pp. 50–61. ISBN: 1-59593-309-3.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[Plo75] Gordon D. Plotkin. “Call-by-Name, Call-by-Value and the λ-Calculus”. In: *Theor. Comput. Sci.* 1.2 (1975), pp. 125–159.

[Plo77] Gordon D. Plotkin. “LCF Considered as a Programming Language”. In: *Theor. Comput. Sci.* 5.3 (1977), pp. 223–255.

[Pop06] Sebastian Pop. “The SSA representation framework: semantics, analyses and GCC implementation”. PhD thesis. École Nationale Supérieure des Mines de Paris, 2006.

[Pro59] Reese T. Prosser. “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*. ACM. 1959, pp. 133–138.

[Rey98] John C. Reynolds. “Definitional Interpreters for Higher-Order Programming Languages”. In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 363–397.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Global Value Numbers and Redundant Computations". In: *POPL '88*. Proceedings of the Fifteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (San Diego, California, USA, Jan. 10–13, 1988). Ed. by Jeanne Ferrante and P. Mager. ACM Press, 1988, pp. 12–27. isbn: 0-89791-252-7.

[San95] André L. M. Santos. "Compilation by Transformation in Non-Strict Functional Languages". PhD thesis. University of Glasgow, 1995.

[SS76] Guy Lewis Steele, Jr. and Gerald Jay Sussman. *Lambda: The Ultimate Imperative.* AI Memo 353. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Mar. 1976.

[Sul+07] Martin Sulzmann et al. "System F with type equality coercions". In: *TLDI '07*. Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation. (Nice, France, Jan. 16, 2007). Ed. by François Pottier and George C. Necula. ACM, 2007, pp. 53–66. isbn: 1-59593-393-X.

[SS75] Gerald Jay Sussman and Guy Lewis Steele, Jr. *Scheme: An interpreter for untyped lambda-calculus.* AI Memo 349. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Dec. 1975.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. "Guarded recursive datatype constructors". In: *POPL 2003*. Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (New Orleans, Louisiana, USA, Jan. 15–17, 2003). Ed. by Alex Aiken and Greg Morrisett. ACM, 2003, pp. 224–235. isbn: 1-58113-628-5.

[XP99] Hongwei Xi and Frank Pfenning. "Dependent Types in Practical Programming". In: *POPL '99*. Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (San Antonio, TX, USA, Jan. 20–22, 1999). Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 214–227. isbn: 1-58113-095-3.