$\lambda^{\bowtie}$: Using Nondeterministic Choice as Part of a Deterministic Calculus

Matthias Blume

Kyoto University
Research Institute for Mathematical Sciences

Abstract

Optimizing compilers for functional languages often use λ-calculus to represent programs during optimization. We investigate the addition of a non-deterministic choice operator to such a language and demonstrate how a compiler can take advantage of it by encoding pending optimization decisions in the calculus itself. We consider two versions of choice (*free* and *constrained*), discuss their differences, and suggest applications for each.

We point out that the resulting system lacks confluence and show how to circumvent this potential problem. To this end, we define what constitutes a non-ambiguous term and argue that non-ambiguity is preserved by our transition rules. This leads to a notion of confluence that, although it is limited to a subset of all expressions, does not rule out any term an actual compiler would be interested in.

Finally, we have a look at how one can construct efficiently computable heuristics for the calculus with choice.

1 Introduction

The construction of optimizing compilers for higher-order languages has benefited from the fact their intermediate representations are modeled after calculi with well-understood semantic properties. A formal semantics for the intermediate language typically provides a very robust framework for deciding which program transformations are correct and which ones would not be.

Many compilers for functional languages such as ML [9, 8] or Haskell [6] use one of the many variations of the λ-calculus as their internal language where (high-level) optimizations are performed. Examples are the CPS language described by Appel [1] that has been used in the SML/NJ implementation of Standard ML [2]. A popular alternative to CPS is A-normal form [13, 5], and recently there is also a continuing trend towards typed intermediate languages [18, 17].

With these tools at hand, writing a *correct* optimizer might not be considered very hard anymore.[1] However, not every correct optimizer is also a *good* optimizer. The theory tells us which transformations are correct, but it helps only very little when trying to *choose* among a set of known-to-be-correct transformation steps.

Program optimization is undecidable. Therefore, an optimizer can do no more than try to approximate and hope that these approximations work reasonably well in practice. This approach typically relies on heuristics that assess the quality of a given approximation. But even with respect to such heuristics, an ultimately positive effect of a transformation step is often not immediately apparent. It can take many additional steps for it to become manifest in an improved estimate.

Without good information to base optimization decisions on, an obvious approach is try out several paths and take the one that in the long run improves the heuristics most. Since the size of the search space quickly explodes, bookkeeping costs for such a technique can become prohibitive unless the search is very carefully pruned. Many compilers simply settle for avoiding deep look-ahead at the expense of perhaps missing some good optimization opportunities.

Aside from bookkeeping difficulties, there is also the problem that an effort of exploring n optimization paths to some large depth seems to be completely wasted if just 2 or 3 steps on the $(n+1)$-st path yield a much better result. We will try to avoid this effect by enabling the compiler to do a *simultaneous* search along all paths at the same time.

To aid the bookkeeping of an aggressive optimizer, this paper recommends a simple device—a *choice* operator—that can easily be added to many λ-calculus-based intermediate representations. In essence, the choice operator moves bookkeeping information about different optimization paths from the meta-level into the calculus itself.

While nondeterministic choice is nothing particularly new, we consider it with the unusual purpose of aiding the optimizer for a *deterministic* language. Therefore, we will put it into very restricted settings where its use is disciplined enough to prevent programs from becoming ambiguous.

1.1 An example

In the untyped λ-calculus[2], using the abbreviations from Figure 2, consider the following expression where A, B, and C are assumed to be arbitrary (perhaps large) subexpressions:

[1] Compiler implementors will certainly disagree with this exaggeration.

[2] see Figures 1 and 11 for a quick reference

$$\begin{array}{rcl} x & \in & \text{Variable} \\ E_\lambda & ::= & x \mid \lambda x.E_\lambda \mid (E_\lambda\ E_\lambda) \\ C_\lambda[\cdot] & ::= & [\cdot] \mid \lambda x.C_\lambda[\cdot] \mid (C_\lambda[\cdot]\ E_\lambda) \mid (E_\lambda\ C_\lambda[\cdot]) \end{array}$$

Figure 1: Syntax of expressions E_λ and contexts $C_\lambda[\cdot]$ in the untyped λ-calculus

$$\begin{array}{rcl} \lambda x_1 x_2 \cdots .M & \stackrel{=}{\scriptscriptstyle def} & \lambda x_1.\lambda x_2 \cdots .M \\ (M_1\ M_2\ M_3\ \cdots\ M_n) & \stackrel{=}{\scriptscriptstyle def} & (\cdots((M_1\ M_2)\ M_3)\cdots\ M_n) \\ \texttt{true} & \stackrel{=}{\scriptscriptstyle def} & \lambda xy.x \\ \texttt{false} & \stackrel{=}{\scriptscriptstyle def} & \lambda xy.y \\ \texttt{not} & \stackrel{=}{\scriptscriptstyle def} & \lambda x.(x\ \texttt{false}\ \texttt{true}) \\ \texttt{xor} & \stackrel{=}{\scriptscriptstyle def} & \lambda xy.(x\ (\texttt{not}\ y)\ y) \\ \texttt{pair} & \stackrel{=}{\scriptscriptstyle def} & \lambda xys.(s\ x\ y) \\ \texttt{fst} & \stackrel{=}{\scriptscriptstyle def} & \lambda p.(p\ \texttt{true}) \\ \texttt{snd} & \stackrel{=}{\scriptscriptstyle def} & \lambda p.(p\ \texttt{false}) \end{array}$$

Figure 2: Common abbreviations for certain λ-expressions.

$$\begin{array}{rcl} x & \in & \text{Variable} \\ E_{\bowtie} & ::= & x \mid \lambda x.E_{\bowtie} \mid (E_{\bowtie}\ E_{\bowtie}) \mid (E_{\bowtie} \bowtie E_{\bowtie}) \\ C_{\bowtie}[\cdot] & ::= & [\cdot] \mid \lambda x.C_{\bowtie}[\cdot] \mid (C_{\bowtie}[\cdot]\ E_{\bowtie}) \mid (E_{\bowtie}\ C_{\bowtie}[\cdot]) \mid (C_{\bowtie}[\cdot] \bowtie E_{\bowtie}) \mid (E_{\bowtie} \bowtie C_{\bowtie}[\cdot]) \end{array}$$

Figure 3: Syntax of expressions $E_{\bowtie}$ and contexts $C_{\bowtie}[\cdot]$ for the untyped λ-calculus with an added operator $\bowtie$ for free choice.

$$x[M/x] = M$$

$$\frac{x \neq y}{y[M/x] = y}$$

$$(L\ M)[N/x] = (L[N/x]\ M[N/x])$$

$$(L \bowtie M)[N/x] = (L[N/x] \bowtie M[N/x])$$

$$(\lambda x.M)[N/x] = \lambda x.M$$

$$\frac{x \neq y \qquad z \text{ is fresh}}{(\lambda y.M)[N/x] = \lambda z.(M[z/y][N/x])}$$

Figure 4: Capture-free substitution.

$$\frac{M[\cdot] = [\cdot]}{M[N] = N} \qquad \frac{L[\cdot] = (M[\cdot]\ N)}{L[O] = (M[O]\ N)}$$

$$\frac{L[\cdot] = \lambda x.M[\cdot]}{L[N] = \lambda x.M[N]} \qquad \frac{L[\cdot] = (M \bowtie N[\cdot])}{L[O] = (M \bowtie N[O])}$$

$$\frac{L[\cdot] = (M\ N[\cdot])}{L[O] = (M\ N[O])} \qquad \frac{L[\cdot] = (M[\cdot] \bowtie N)}{L[O] = (M[O] \bowtie N)}$$

Figure 5: Context substitution.

$$(\lambda x.M\ N) \rightarrow_\lambda M[N/x] \qquad (\text{Beta}_\lambda)$$

$$\frac{N \rightarrow_\lambda N'}{M[N] \rightarrow_\lambda M[N']} \qquad (\text{Context}_\lambda)$$

$$M \rightarrow_\lambda{}^* M \qquad (\text{Null}_\lambda)$$

$$\frac{M \rightarrow_\lambda M' \qquad M' \rightarrow_\lambda{}^* M''}{M \rightarrow_\lambda{}^* M''} \qquad (\text{Trans}_\lambda)$$

Figure 6: Transition relation for the original λ-calculus. Capture-free substitution and context substitution operations are those shown in Figure 4 and 5 (but restricted to λ-terms without choice).

$$(\lambda x.M\ N) \rightarrow_{\bowtie} M[N/x] \qquad (\text{Beta})$$

$$\frac{N \rightarrow_{\bowtie} N'}{M[N] \rightarrow_{\bowtie} M[N']} \qquad (\text{Context})$$

$$(L \bowtie (M \bowtie N)) \rightarrow_{\bowtie} ((L \bowtie M) \bowtie N) \qquad (\text{Ch-Assoc})$$

$$(M \bowtie N) \rightarrow_{\bowtie} (M \bowtie N) \qquad (\text{Ch-Symm})$$

$$\lambda x.(M \bowtie N) \rightarrow_{\bowtie} (\lambda x.M \bowtie \lambda x.N) \qquad (\text{Lam-Dist})$$

$$((L \bowtie M)\ N) \rightarrow_{\bowtie} ((L\ N) \bowtie (M\ N)) \qquad (\text{App-Dist-L})$$

$$(L\ (M \bowtie N)) \rightarrow_{\bowtie} ((L\ M) \bowtie (L\ N)) \qquad (\text{App-Dist-R})$$

$$(M \bowtie N) \rightarrow_{\bowtie} M \qquad (\text{Choose})$$

$$M \rightarrow_{\bowtie}{}^* M \qquad (\text{Null})$$

$$\frac{M \rightarrow_{\bowtie} M' \qquad M' \rightarrow_{\bowtie}{}^* M''}{M \rightarrow_{\bowtie}{}^* M''} \qquad (\text{Trans})$$

Figure 7: Transition relation for the calculus with choice $\bowtie$.

$$\begin{array}{ll}
1 & (\lambda f.(\texttt{pair} \\
2 & \quad\quad (\texttt{pair}\ (g\ f) \\
3 & \quad\quad\quad\quad (\lambda y.(y\ (\texttt{pair}\,\texttt{true}\ A))\ f)) \\
4 & \quad\quad (\lambda z.(z\ (\texttt{pair}\ w\ B))\ f)) \\
5 & \lambda x.((\texttt{fst}\ x)\ (\texttt{fst}\ x)\ (C\ (\texttt{snd}\ x))))
\end{array}$$

While an evaluator would have no choice but to eventually β-reduce the outermost application, the optimizer of a compiler might not indiscriminately want to do so. The body of the abstraction that binds f contains three separate occurences of f. Thus, a β-reduction would triplicate the argument expression. But C might be so large that the compiler's heuristics do not consider it worthwhile to even just try.

Now let us add a new operator to the calculus as shown in Figure 3. Intuitively, the meaning of an expression $(M \bowtie N)$ is the non-deterministic choice between evaluating M and evaluating N. Guided by this intuition, we want the choice operator to have various useful properties such as being symmetric and associative. Moreover, both abstraction as well as application should be distributive with respect to choice. A transitions system for this calculus is given in Figure 7. Notice that this transition system is identical to that of the underlying λ-calculus without choice as far as expressions are concerned where $\bowtie$ does not occur.

With this preparation, we can come back to our example. Let us postulate a modified version of the Beta rule that states

$$(\lambda x.M\ N) \to (\lambda y.(M[(y \bowtie N)/x])\ N) \qquad (\textsc{Beta}_{\bowtie})$$

where y is a fresh variable.

Now have a closer look at each of the three occurences of f (on lines 2, 3, and 4 in the example) if this new rule is applied to the outermost redex. (For brevity, we write $\lambda x.F\{x\}$ for the argument of this outermost application and also ignore the α-renaming of f—assuming f does not occur free in $F\{x\}$.)

On line 2, the application $(g\ f)$ undergoes the following transition steps:

$$\begin{array}{rl}
 & (g\ (f \bowtie \lambda x.F\{x\})) \\
\xrightarrow[\bowtie]{\text{by App-Dist-R}} & ((g\ f) \bowtie (g\ \lambda x.F\{x\})) \\
\xrightarrow[\bowtie]{\text{by Choose}} & (g\ f)
\end{array}$$

Note that the choice on the last line can be made either way, but most compilers would want to chose the one shown here.

On line 3, the original application is distributed over the choice in a similar fashion. One of the alternatives then quickly arrives at $(f\ (\texttt{pair}\,\texttt{true}\ A))$, while the other undergoes the following sequence of transitions:

$$\begin{array}{rl}
 & (\lambda y.(y\ (\texttt{pair}\,\texttt{true}\ A))\ \lambda x.F\{x\}) \\
\to_{\bowtie} & (\lambda x.F\{x\}\ (\texttt{pair}\,\texttt{true}\ A)) \\
 & \vdots \\
\to_{\bowtie} & (\texttt{true}\,\texttt{true}\ (C\ A)) \\
\to_{\bowtie} & \texttt{true}
\end{array}$$

Clearly, in this case the compiler's heuristics should favor the right-hand operand of $\bowtie$ (i.e., a Ch-Symm transition before applying Choose).

Line 4 looks very similar to line 3, except `true` is replaced with a free variable w. Thus, none of the reductions that depended on `true` can be performed. Like in the previous case, transformations eventually produce a choice between two expressions. One of them is $(f\ (\texttt{pair}\ w\ B))$, the other one is produced by this transformation sequence:

$$\begin{array}{rl}
 & (\lambda z.(z\ (\texttt{pair}\ w\ B))\ \lambda x.F\{x\}) \\
\to_{\bowtie} & (\lambda x.F\{x\}\ (\texttt{pair}\ w\ B)) \\
 & \vdots \\
\to_{\bowtie} & (w\ w\ (C\ B))
\end{array}$$

Thus, either side of the final choice still contains B. Moreover, the second one also contains C. C is still present in the original $F\{x\}$ on line 5, i.e., the "definition" of f. This is necessary since on line 2 we have settled for keeping the original expression $(g\ f)$, the variable f continues to occur and its definition cannot be dropped. Considering all this, an optimizer's heuristics would probably decide to keep the first alternative, namely $(f\ (\texttt{pair}\ w\ B))$, and discard the second. The final overall expression therefore is

$$\begin{array}{ll}
1 & (\lambda f.(\texttt{pair} \\
2 & \quad\quad (\texttt{pair}\ (g\ f) \\
3 & \quad\quad\quad\quad \texttt{true}) \\
4 & \quad\quad (f\ (\texttt{pair}\ w\ B))) \\
5 & \lambda x.((\texttt{fst}\ x)\ (\texttt{fst}\ x)\ (C\ (\texttt{snd}\ x))))
\end{array}$$

which would not have been reachable with the choice-free version of the calculus.

1.2 The purpose of choice

As demonstrated in this example, the purpose of the choice operator is to delay the effect of a reduction and to maintain—as part of the calculus—the alternatives of both reducing and not reducing. The Choose rule always provides the possibility of finalizing pending choices by arbitrarily favoring one side. Moreover, a pending choice can be propagated to several different places and local knowledge at those places can later be used to actually make the decision at each instance.

An expression with embedded choices represents a "snapshot" of a heuristic breadth-first search in optimization space. Since this space is potentially very big, expressions will have the tendency to grow quickly. Thus, the Choose rule must be applied often, and the heuristics for deciding when and how will have to be similar to the heuristics in "traditional" optimizers. When viewed from this angle, it becomes clear that $\bowtie$ is certainly no silver bullet.

On the other hand, since the choice-based optimizer essentially performs a breadth-first instead of a depth-first search, it is easier to trade precision in one search branch for precision in another. A branch with a pending choice that "looks promising" can be explored arbitrarily far as long as this is payed for by pruning other, less promising branches. To implement such behavior in an optimizer for a language without choice is harder because there it is difficult to compare different branches "side-by-side".

2 Free choice

The choice operator $\bowtie$ that we have seen in the previous section does not constrain the use of other transformation

$$\mathcal{C}[\![x]\!] = \{x\}$$
$$\mathcal{C}[\![\lambda x.M]\!] = \{\lambda x.M' \mid M' \in \mathcal{C}[\![M]\!]\}$$
$$\mathcal{C}[\![(M\ N)]\!] = \{(M'\ N') \mid M' \in \mathcal{C}[\![M]\!], N' \in \mathcal{C}[\![N]\!]\}$$
$$\mathcal{C}[\![(M \bowtie N)]\!] = \mathcal{C}[\![M]\!] \cup \mathcal{C}[\![N]\!]$$

Figure 8: Calculating the set of *commits* $\mathcal{C}[\![M]\!]$ of a $\lambda^{\bowtie}$-expression M.

rules. We will call such an non-constraining choice operator a *free choice*. (Section 5 discusses a *constrained* variety that can also sometimes be useful.)

There is no technical reason for not using typed calculi, but their formal treatment is more elaborate. There are more expression formation rules (pairing, projection, recursion, etc.), so there will have to be many more rules that describe how $\bowtie$ interacts with these constructs—even though the treatment turns out to be analogous in each case. For the sake of economy of notation we will thus mainly consider the untyped λ-calculus with choice (henceforth called the $\lambda^{\bowtie}$-calculus) and only briefly mention typed versions where types are relevant to the discussion.

2.1 Non-confluence

The immediate trouble with the introduction of choice into a λ-calculus is that the resulting system is no longer *confluent*, meaning that there are expressions L, M, and N with $L \rightarrow_{\bowtie} M$ and $L \rightarrow_{\bowtie} N$ but there is no O such that $M \rightarrow_{\bowtie}^* O$ and $N \rightarrow_{\bowtie}^* O$. In fact, examples for non-confluence can be trivially constructed:

$$(\texttt{true} \bowtie \texttt{false})$$

Clearly, we must avoid the possibility of different transformations to arrive at programs with different meanings. Therefore, we will restrict the calculus to "well-behaved" expressions, i.e., those that restore the Church-Rosser property (confluence) of the calculus. How this is to be achieved will be our topic for the remainder of this section.

2.2 Commits

Consider a $\lambda^{\bowtie}$-expression M. We define the set of *commits* $\mathcal{C}[\![M]\!] \in E$ as the set of expressions that can be derived from M by deciding each occurrence of $\bowtie$ one way or the other (Figure 8).

$\mathcal{C}[\![M]\!]$ has the following properties:

- If M is an ordinary λ-term (i.e., does not contain $\bowtie$), then $\mathcal{C}[\![M]\!] = \{M\}$.
- Each $M' \in \mathcal{C}[\![M]\!]$ can be obtained from M by only applying two rules: Choose and Context.
- By only applying symmetry-, associativity-, and distribution rules (Ch-Symm, Ch-Assoc, Lam-Dist, App-Dist-L, App-Dist-R), one can for each M derive a "canonical" expression $((\ldots(N_1 \bowtie N_2) \bowtie \ldots N_{k-1}) \bowtie N_k)$ so that $\mathcal{C}[\![M]\!] = \{N_1, N_2, \ldots, N_{k-1}, N_k\}$. In other words, M can be evolved into an expression that simultaneously represents all members of $\mathcal{C}[\![M]\!]$ in a syntactic way. There can be duplicates of the form $N_i = N_j$ for some $1 \le i, j \le k$, though.

It may be tempting to tackle the ambiguity problem by postulating that an expression M be well-formed whenever $\mathcal{C}[\![M]\!]$ contains pairwise equivalent expressions only.[3] We call expressions that fail this test *immediately inconsistent*.

$$M \text{ immediately inconsistent} \overset{=}{\underset{def}{}} \\ \nexists N : \forall M' \in \mathcal{C}[\![M]\!] : M' \rightarrow_{\lambda}^* N$$

Obviously, avoiding immediate inconsistency is necessary. However, for our purposes it is not sufficient. The first (and lesser) problem is that the property of being immediately inconsistent is not recursively enumerable. This would not be fatal if its negation (i.e., *not* being immediately inconsistent—a property that is recursively enumerable) would be preserved by $\rightarrow_{\bowtie}$. Unfortunately, this is not the case. Consider:

$$(\lambda x.(\texttt{xor}\ x\ x)\ (\texttt{true} \bowtie \texttt{false}))$$

This expression has two commits that both have the same normal form `false`. Therefore, the expression is not immediately inconsistent even though a simple Beta step turns it into something that is:

$$(\texttt{xor}\ (\texttt{true} \bowtie \texttt{false})\ (\texttt{true} \bowtie \texttt{false}))$$

Two of the four commits of this new expression can be transformed to `true`, the other two to `false`, and there is no N so that $\texttt{true} \rightarrow_{\lambda}^* N$ and also $\texttt{false} \rightarrow_{\lambda}^* N$.

2.3 Strong non-ambiguity

What we are looking for is a property that rules out immediate inconsistency and which will also be preserved by program transformations. At a minimum, the $\rightarrow_{\bowtie}$ relation, i.e., each rule in Figure 7 must preserve it.

The trouble with the previous counterexample can be attributed to the fact that there is a subterm which—in some sense—is inconsistent but for which the difference in meaning happens to cancel with itself. After the Beta step, the two incompatible sides meet each other directly, and the original cancellation of their difference no longer occurs.

One way to ensure preservation of equivalent commits (implying, as we will see, confluence) is to rule out inconsistent subterms. However, simply requiring that all subterms also not be immediately inconsistent is not helpful. To see why, consider applying $\text{Beta}_{\bowtie}$ to

$$(\lambda x.x\ y)$$

The result is

$$(\lambda z.(z \bowtie y)\ y)$$

which now has a subterm $(z \bowtie y)$ that is immediately inconsistent—although the intention was to treat z and y as synonyms inside the abstraction. Thus, the correct way of defining what constitutes a well-formed $\lambda^{\bowtie}$-expression must take binding context into account.

Let the finite map $\Gamma \in \text{Variable} \rightarrow E_{\lambda}$ be a *binding environment* and $A \in E_{\lambda}{}^*$ a (possibly empty) sequence of E_{λ}-terms $\langle M_1, \ldots, M_k \rangle$. As usual, $\Gamma[x \mapsto M]$ denotes

[3] As usual, $M, N \in E_{\lambda}$ are considered equivalent if there is some O such that $M \rightarrow_{\lambda}^* O \wedge N \rightarrow_{\lambda}^* O$.

$$\frac{\Gamma, A \vdash M \rhd N \qquad N \rightarrow_\lambda O}{\Gamma, A \vdash M \rhd O} \quad \text{(SUB)}$$

$$\frac{x \notin \mathrm{dom}(\Gamma)}{\Gamma, A \vdash x \rhd x} \quad \text{(FREE)}$$

$$\frac{\Gamma(x) = M}{\Gamma, A \vdash x \rhd M} \quad \text{(BOUND)}$$

$$\frac{\begin{array}{c}\Gamma, \langle\rangle \vdash N \rhd N' \\ \Gamma, \langle A \mid N' \rangle \vdash M \rhd M'\end{array}}{\Gamma, A \vdash (M\ N) \rhd M'} \quad \text{(CHARGE)}$$

$$\frac{\Gamma[x \mapsto N], A \vdash M \rhd O}{\Gamma, \langle A \mid N \rangle \vdash \lambda x.M \rhd O} \quad \text{(DISCHARGE)}$$

$$\frac{y \ \text{ is fresh} \qquad \Gamma[x \mapsto y], \langle\rangle \vdash M \rhd O}{\Gamma, \langle\rangle \vdash \lambda x.M \rhd \lambda y.O} \quad \text{(ALPHA)}$$

$$\frac{\Gamma, A \vdash M \rhd O \qquad \Gamma, A \vdash N \rhd O}{\Gamma, A \vdash (M \bowtie N) \rhd O} \quad \text{(CONS)}$$

Figure 9: Inference rules for judgements of the form $\Gamma, A \vdash M \rhd N$. These judgements are pronounced: "In Γ with A, M consistently reduces to N.

the environment that maps x to M and otherwise coincides with Γ. Furthermore, for any $A = \langle M_1, \ldots, M_k \rangle$, the sequence $\langle M_1, \ldots, M_k, M \rangle$ can be written as $\langle A \mid M \rangle$. For $\lambda^{\bowtie}$-expressions M and λ-expressions N we then define the relation $\Gamma, A \vdash M \rhd N$ which should be pronounced as "in Γ with A, M consistently reduces to N" as shown in Figure 9. *Strong non-ambiguity* is then defined as follows:

Definition 1 *An expression $M \in E_{\bowtie}$ is called strongly non-ambiguous if and only if there is some expression $N \in E_\lambda$ such that $\emptyset, \langle\rangle \vdash M \rhd N$.*

We write $\vdash M \rhd N$ as an abbreviation for $\emptyset, \langle\rangle \vdash M \rhd N$, so strong non-ambiguity of M can be concisely expressed as $\exists N : (\vdash M \rhd N)$. Note that from $\vdash M \rhd N$ follows $M \rightarrow_{\bowtie}{}^* N$.

Strong non-ambiguity also implies that the term in question cannot be immediately inconsistent:

Proposition 1 *If $\vdash M \rhd N$, then $\exists N' : \forall M' \in \mathcal{C}[\![M]\!] : M' \rightarrow_\lambda{}^* N'$.*

The proof of Proposition 1 proceeds by straightforward induction on the structure of the expression.

In the following discussion we will drop the qualifier "strong" and refer to strong non-ambiguity (as opposed to *weak* non-ambiguity—the topic of section 2.8) simply as "non-ambiguity".

2.4 Preservation of non-ambiguity

To show that an expression involving choice is non-ambiguous, one must be able to prove that two expressions M and N are equivalent, i.e., that there is an expression O that can be reached from both M and N. Of course, there is no general decision procedure for this because it would require solving the halting problem. However, non-ambiguous expressions can be enumerated, and since a compiler always starts with a non-ambiguous expression (if the source language does not have a $\bowtie$ operator), it is enough to make sure that non-ambiguity is preserved by the transition relation.

Proposition 2 *If $\vdash M \rhd M'$ and $M \rightarrow_{\bowtie} N$, then $\exists N' : (\vdash N \rhd N' \wedge M' \rightarrow_\lambda{}^* N')$.*

To prove Proposition 2 we show that the slightly stronger version holds where Γ and A are not necessarily empty: if $\Gamma, A \vdash M \rhd M'$ and $M \rightarrow_{\bowtie} N$, then $\exists N' : (\Gamma, A \vdash N \rhd N' \wedge M' \rightarrow_\lambda{}^* N')$. For this we proceed by cases where each transition rule is considered in turn. The individual cases use simultaneous structural induction on M and N. A slight technical difficulty is that the structures of M and N are not completely identical. For example, in the case of the BETA rule even the binding structures are different. To be able to use the induction hypothesis, environments Γ and the list of pending applications A must be "fudged" according to Lemma 1 to look the same for both M and N.

Notice that the substitution that occurs during a BETA-transition is mimicked by the interplay of CHARGE-, DISCHARGE, ALPHA-, and BOUND rules. The rôles of Γ and A in judgments of the form $\Gamma, A \vdash M \rhd N$ is the following: Γ remembers pending substitutions and A records pending applications of the BETA transition rule. As a result, the judgments treat β-redexes as if they have already been reduced and, thus, do not distinguish between expressions before or after an application of the BETA rule.

The following lemma justifies the "fudging" of environments and application lists we just mentioned:

Lemma 1 *If $\Gamma, A \vdash M \rhd N$ and x does not occur free in M, then $\Gamma, \langle A \mid Y \rangle \vdash \lambda x.M \rhd N$ for arbitrary expressions Y.*

Proof: Straightforward by induction on the structure of M.

2.5 Confluence

From Propositions 1 and 2 one can conclude that for non-ambiguous terms, $\rightarrow_{\bowtie}$ keeps the sets of commits consistent. This means, that the absence of immediate inconsistency is also preserved:

Proposition 3 *If $\vdash M \rhd M'$ and $M \rightarrow_{\bowtie} N$, then $\exists N' : \forall L \in \mathcal{C}[\![M]\!] \cup \mathcal{C}[\![N]\!] : L \rightarrow_{\bowtie}{}^* N'$.*

From this we conclude that $\rightarrow_{\bowtie}$ is indeed confluent for non-ambiguous expressions:

Proposition 4 *If $\vdash M \rhd N$ and there are terms $M', M'' \in E_{\bowtie}$ such that $M \rightarrow_{\bowtie} M'$, and $M \rightarrow_{\bowtie} M''$, then $\exists N' : N \rightarrow_\lambda{}^* N' \wedge M' \rightarrow_{\bowtie}{}^* N' \wedge M'' \rightarrow_{\bowtie}{}^* N'$.*

Proof: First we use induction on the number of $\rightarrow_{\bowtie}$ steps in a given $\rightarrow_{\bowtie}{}^*$ transition sequence to show that Proposition 3 extends to $\rightarrow_{\bowtie}{}^*$. Now consider M, N such that $\vdash M \rhd N$. We pick an arbitrary element $M_0 \in \mathcal{C}[\![M]\!]$. For M' with $M \rightarrow_{\bowtie} M'$ we then pick $M_0' \in \mathcal{C}[\![M']\!]$ in a similar way and note that since M_0' can be obtained from M' by using CHOOSE we have $M' \rightarrow_{\bowtie}{}^* M_0'$ and therefore $M \rightarrow_{\bowtie}{}^* M_0'$. By the extension of Proposition 3 we conclude that $\exists \hat{M}' : M_0 \rightarrow_{\bowtie}{}^* \hat{M}' \wedge M_0' \rightarrow_{\bowtie}{}^* \hat{M}'$. However, since both M_0 and M_0' are choice-free expressions, we even have $M_0 \rightarrow_\lambda{}^* \hat{M}'$ and $M_0' \rightarrow_\lambda{}^* \hat{M}'$. By the same reasoning for an M'' with $M \rightarrow_{\bowtie} M''$ we get some $\hat{M}''$ so that $M_0 \rightarrow_\lambda{}^* \hat{M}''$ and $M_0'' \rightarrow_\lambda{}^* \hat{M}''$. We also know that $M \rightarrow_{\bowtie}{}^* N$ and that N itself is choice-free. Therefore, there is an $\hat{N}$ with $M_0 \rightarrow_\lambda{}^* \hat{N}$ and $N \rightarrow_\lambda{}^* \hat{N}$. Since $\rightarrow_\lambda{}^*$ is already known to be confluent, we conclude that there must be an N' with $N \rightarrow_\lambda{}^* N' \wedge M_0' \rightarrow_\lambda{}^* N' \wedge M_0'' \rightarrow_\lambda{}^* N'$ and therefore also $N \rightarrow_\lambda{}^* N' \wedge M' \rightarrow_{\bowtie}{}^* N' \wedge M'' \rightarrow_{\bowtie}{}^* N'$. $\square$

2.6 Choice-free terms

Proposition 5 *Every choice-free term $M \in E_\lambda$ is (strongly) non-ambiguous.*

The proof for this proceeds by induction on the structure of M, showing that $\Gamma, A \vdash M \rhd M$.

2.7 Adding rules

In a real compiler, the initial expression will typically be an ordinary λ-term without choice.[4] According to Proposition 5, choice-free expressions are non-ambiguous. However, the notion of non-ambiguity was introduced to have a handle on confluence, and choice-free terms are always confluent because for them $\rightarrow_{\bowtie}$ coincides with $\rightarrow_\lambda$.

This coincidence is caused by the lack of choice-introducing rules for $\rightarrow_{\bowtie}$. Thus, so far we still have nothing more than a *framework* relative to which we must now consider additional rules that introduce choice. Here, the one rule we are interested in is $\text{Beta}_{\bowtie}$ (i.e., the one that we have already seen when we discussed the introductory example).

To justify that an added rule does not disturb confluence, one must show that it also preserves non-ambiguity:

Proposition 6 *If $\vdash M \rhd M'$ and $M \xrightarrow{by\ \text{Beta}_{\bowtie}}_{\bowtie} N$, then $\exists N'$: $(\vdash N \rhd N' \wedge M' \rightarrow_\lambda{}^* N')$.*

The proof for Proposition 6 is very similar to that of the Beta case in the proof for Proposition 2, the main difference being that here the β-redex does not disappear (but its bound variable gets renamed).

2.8 Weak non-ambiguity

The rules in Figure 9 mandate for $\Gamma, A \vdash (M\ N) \rhd X$ to hold that one must be able to show $\Gamma, A \vdash N \rhd Y$ for some Y. This requirement can occasionally be weakened because M might discard its argument and therefore would not care whether it was ambiguous or not.

The notion of *weak non-ambiguity* accounts for this by using modified Charge- and Bound rules. Figure 10 gives the rules for judgments of the form $\Gamma, A \vdash M \rhd' N$. Weak non-ambiguity is then defined analogous to its strong counterpart (see Definition 1):

Definition 2 *An expression $M \in E_{\bowtie}$ is called weakly non-ambiguous if and only if there is some expression $N \in E_\lambda$ such that $\emptyset, \langle\rangle \vdash M \rhd' N$.*

Weak non-ambiguity is adequate for a λ-calculus with call-by-name semantics (see Figure 11). However, it is *not* adequate for a calculus with call-by-value semantics (Figure 12) because, for example,

$$((\lambda x.y)\ (\mathtt{true} \bowtie (\lambda x.(x\ x)\ \lambda x.(x\ x))))$$

is weakly non-ambiguous, but its two commits do not have the same meaning (one reduces to y, the other one diverges).

Notice, however, that from a practical point of view, there does not seem to be an advantage of using weak non-ambiguity—even under call-by-name semantics—because so far we have not encountered a transition that an optimizer would want to use and that preserves weak but not strong non-ambiguity.

[4] An exception to this is discussed in section 3.

$$\frac{\Gamma, A \vdash M \rhd' N \qquad N \rightarrow_\lambda O}{\Gamma, A \vdash M \rhd' O} \quad (\text{Sub'})$$

$$\frac{x \notin \text{dom}(\Gamma)}{\Gamma, A \vdash x \rhd' x} \quad (\text{Free'})$$

$$\frac{\Gamma(x) = M \qquad \Gamma, A \vdash M \rhd' M'}{\Gamma, A \vdash x \rhd' M'} \quad (\text{Bound'})$$

$$\frac{\Gamma, \langle A \mid N \rangle \vdash M \rhd' M'}{\Gamma, A \vdash (M\ N) \rhd' M'} \quad (\text{Charge'})$$

$$\frac{\Gamma[x \mapsto N], A \vdash M \rhd' O}{\Gamma, \langle A \mid N \rangle \vdash \lambda x.M \rhd' O} \quad (\text{Discharge'})$$

$$\frac{y \text{ is fresh} \qquad \Gamma[x \mapsto y], \langle\rangle \vdash M \rhd' O}{\Gamma, \langle\rangle \vdash \lambda x.M \rhd' \lambda y.O} \quad (\text{Alpha'})$$

$$\frac{\Gamma, A \vdash M \rhd' O \qquad \Gamma, A \vdash N \rhd' O}{\Gamma, A \vdash (M \bowtie N) \rhd' O} \quad (\text{Cons'})$$

Figure 10: Inference rules for judgements of weak consistency which have the form $\Gamma, A \vdash M \rhd' N$. (For comparison with the strong version, see Figure 9.)

3 A real-world example: Cross-module inlining

The SML/NJ compiler [2] represents compilation units as closed functions that map imported values to exported values [3]. This fact can be exploited to conveniently express cross-module inlining by manipulating the λ-terms that represent these closed functions in intermediate language [4].

If the source code of compilation unit U_2 has a free reference to a variable x exported from U_1, then F_2—the function representing U_2—will have a formal parameter x and the linker will make sure that the value for x as produced by F_1 (the function representing U_1) will be passed to F_2.

The λ-splitting transformation modifies the code for F_2 by attaching code that originally came from F_1. In particular, let X be some subterm of F_1 that (in its original context) produces the value of x. For simplicity, let us assume that x was the only argument of F_2 and that X has only one free variable y. Then λ-splitting would turn F_2 into

$$\lambda y.(F_2\ X)$$

and let the existing intra-modular optimizer take care of the rest, i.e., let it propagate the expression X to all free occurrences of variable x in the body of F_2 by reducing the outermost β-redex. Notice that F_1 must also be modified to export y if it wasn't exported already, and the linker must be made aware of these changes so it can pass correct values.

This works well if X is small enough for the optimizer's heuristics to actually enable the β-reduction. If the reduction does not occur, then nothing is won and one suddenly has a residual copy of X in the code for compilation unit U_2. (Of course, if the reduction *does* occur, there might be even more residual copies of X in F_2—which is precisely why the optimizer would sometimes decide to hold off the reduction of the outermost β-redex.)

In real programs, there is usually more than one importing U_2 for an exporting U_1, so X might be copied often. Moreover, it is frequently the case that the code for F_1 itself continues to calculate the value of x anyway (because it is needed in other parts of U_1). Therefore, residual copies of X

$$x \Downarrow x \qquad (\textsc{Var})$$

$$\lambda x.M \Downarrow \lambda x.M \qquad (\textsc{Lam})$$

$$\frac{M \Downarrow \lambda x.B \qquad B[N/x] \Downarrow V}{(M\ N) \Downarrow V} \qquad (\textsc{App})$$

Figure 11: Classical natural ("big step") semantics for the call-by-name λ-calculus.

$$x \Downarrow' x \qquad (\textsc{Var'})$$

$$\lambda x.M \Downarrow' \lambda x.M \qquad (\textsc{Lam'})$$

$$\frac{M \Downarrow' \lambda x.B \qquad N \Downarrow' N' \qquad B[N'/x] \Downarrow' V}{(M\ N) \Downarrow' V} \qquad (\textsc{App'})$$

Figure 12: Classical natural semantics for the call-by-value λ-calculus.

$$\frac{M[N/x]_p = M' \qquad y \text{ is fresh}}{(\lambda x.M\ N) \to (\lambda y.M'[y/x]\ N)} \qquad (\textsc{Beta}^1_p)$$

$$\frac{\nexists M' : M[N/x]_p = M'}{(\lambda x.M\ N) \to M} \qquad (\textsc{Beta}^2_p)$$

$$\frac{N \to N'}{M[N] \to M[N']} \qquad (\textsc{Context}_p)$$

Figure 13: Transition rules for a λ-calculus with partial β-reduction. Notice that antecedents of the form $M[N/x]_p = M'$ (see Figure 14) double as tests for the existence of free occurences of x in M.

$$x[M/x]_p = M$$

$$\frac{L[N/x]_p = L'}{(L\ M)[N/x]_p = (L'\ M)}$$

$$\frac{M[N/x]_p = M'}{(L\ M)[N/x]_p = (L\ M')}$$

$$\frac{x \neq y \qquad z \text{ is fresh} \qquad M[z/y][N/x]_p = M'}{(\lambda y.M)[N/x]_p = \lambda z.M'}$$

Figure 14: Capture-free, partial substitution. An equation of the form $M[N/x]_p = M'$ can be derived only if x occurs free in M. Note that in the last rule the α-renaming substitution of z for y in M is not partial.

in the code for compilation units other than U_1 are almost always redundant.

Adding choice to the intermediate language can solve this problem.[5] Instead of turning F_2 into $\lambda y.(F_2\ X)$, one can turn it into

$$\lambda xy.(F_2\ (x \bowtie X))$$

Of course, the linker must now pass both x and y to the resulting code for U_2.

A minor technical difficulty related to this linkage requirement arises here because this code is not strongly non-ambiguous. (It could be weakly non-ambiguous, but in interesting cases not even that is the case.) However, one can show that there is some Y for which

$$\emptyset, \langle y, X\rangle \vdash \lambda xy.(F_2\ (x \bowtie X)) \rhd Y$$

The presence of a "pre-charged" list of pending applications corresponds to the fact that for λ-splitting to be correct, the linker must pass correct values to compilation units. Without choice, a broken linker will likely cause incorrectly instantiated compilation units, but the resulting (wrong) semantics will be independent of what the intra-module optimizer did. With choice, this is no longer the case: different optimizations lead to different semantics. Of course, this is not a problem because we are not interested in the case of a broken linker.

4 Designing choice-introduction rules

A rule of thumb for designing new choice-introducing rules is to look at places where the original calculus leaves some freedom for decision-making. A new rule can usually be designed by taking two of the alternatives, joining them via $\bowtie$, and then *inversely* applying some of the distribution laws.

[5] In fact, the thought of adding choice first came up during the work on λ-splitting.

To illustrate this point, let us have a look at the design of the $\textsc{Beta}_{\bowtie}$ rule. Finding the connection that we are seeking requires some work in this case because at first inspection there is no rule in the original calculus whose application and non-application form the choice-pair of the $\textsc{Beta}_{\bowtie}$ rule. In fact, this should come as no surprise since using the new rule one can reach expressions that were not reachable without it. A simple example is the following:

$$\begin{array}{ll} & (\lambda x.(x\ x)\ y) \\ \xrightarrow{\text{by } \textsc{Beta}_{\bowtie}}_{\bowtie} & (\lambda z((z \bowtie y)\ (z \bowtie y))\ y) \\ \to_{\bowtie}{}^* & (\lambda z(z\ y)\ y) \end{array}$$

The last expression is not reachable by only using $\to_{\bowtie}$ because it constitutes the result of a "partial" application of the $\textsc{Beta}$ rule. But notice that even though the expression itself is not reachable, one can always let another application of $\textsc{Beta}$ follow—which then hides the effect of the earlier "partial" application.

The problem is that the original $\textsc{Beta}$ rule performs changes in more than one place simultaneously. Although it requires some tedium, one can modify the λ-calculus by replacing $\textsc{Beta}$ with a partial rule (see Figure 13) that keeps the effect of each substitution localized. The resulting calculus has the same confluence properties as the original calculus although the new one can reach some intermediate expressions that were not reachable before.

Now, if $\textsc{Beta}^1_p$ is applicable to $(\lambda x.M\ N)$ (which is to say that M has at least one free occurrence of x), then placing the result of this application next to the original expression (the "result" of the rule's non-application) yields two expression that up to α-conversion differ only in the place where the partial substitution replaced x with N. Thus, one could join the two expressions with $\bowtie$, then "push" the $\bowtie$ back to that original occurrence of x, and arrive at the $\lambda^{\bowtie}$-variant of the $\textsc{Beta}^1_p$ rule.

Finally, we observe that in the $\lambda^{\bowtie}$-calculus one can undo the introduction of partial β-reduction: full $\text{B}\textsc{eta}_{\bowtie}$ can be obtained from the partial version by using sequences of $\text{B}\textsc{eta}_p^1$ followed by one $\text{B}\textsc{eta}_p^2$ step, while the partial rule can be obtained from one $\text{B}\textsc{eta}_{\bowtie}$ step followed by several applications of Choose.

5 Constrained choice

As we have seen, studying the mechanics of a traditional optimizer's decision-making can guide the design of rules that introduce choice. However, there are cases where this intuition breaks down and a legal program transformation would lead to an invalid introduction rule. Consider the following fragment of a program written in Standard ML:

```
structure F :> sig
    type t
    val f :  t
    val a :  t -> int * int -> int
end = struct
    type t = int * int -> int
    fun f (x, y) = ...
    fun a f (y, x) = f (x, y)
end
```

In this example, an ML compiler can apply the following reasoning:

- Type `t` is abstract (as indicated by the *opaque* signature match written using "`:>`").
- Therefore, there is no way of directly invoking any function of type `t` from outside structure `F`.
- Function `f` is such a function of type `t`.
- Therefore, one can rewrite `f` (together with all other functions of type `t`) to use a more efficient calling convention.
- In particular, one can reverse the order of the arguments and modify `a` accordingly (as well as all other call sites of `f` inside structure `F`).

Here is what the resulting code would look like. Notice how `a` was simplified:

```
structure F' :> sig
    type t
    val f :  t
    val a :  t -> int * int -> int
end = struct
    type t = int * int -> int
    fun f (y, x) = ...
    fun a f = f
end
```

Obviously, there is freedom in the decision of whether or not to apply such a transformation. But to think that this freedom justifies a corresponding introduction rule for $\bowtie$ is wrong. Consider a client module that invokes `f` via `a`. If the client had *free* choice between `F` and `F'`, then it would be translated as something like:

```
(F⋈F').a ((F⋈F').f)
```

In cases where the two choices are decided differently, this code would match `F.a` with `F'.f` (or vice versa), which changes the semantics of the program.

$$
\begin{array}{rcl}
x & \in & \text{Variable} \\
E_\diamond & ::= & x \mid \lambda x.E_\diamond \mid (E_\diamond\, E_\diamond) \mid (E_\diamond \bowtie E_\diamond) \mid (E_\diamond \diamond E_\diamond) \\
C_\diamond[\cdot] & ::= & [\cdot] \mid \lambda x.C_\diamond[\cdot] \mid (C_\diamond[\cdot]\, E_\diamond) \mid (E_\diamond\, C_\diamond[\cdot]) \mid \\
 & & (C_\diamond[\cdot] \bowtie E_\diamond) \mid (E_\diamond \bowtie C_\diamond[\cdot]) \mid \\
 & & (C_\diamond[\cdot] \diamond E_\diamond) \mid (E_\diamond \diamond C_\diamond[\cdot])
\end{array}
$$

Figure 15: Syntax of expressions $E_\diamond$ and contexts $C_\diamond[\cdot]$ for the untyped λ-calculus with both free choice $\bowtie$ and constrained choice $\diamond$.

5.1 Avoiding too much freedom

Evidently there is too much freedom of choice here because two different concrete realizations of the same abstract type can "meet" each other. To avoid this problem, let us first introduce a new operator $\diamond$ which we refer to as *constrained* choice (see Figure 15). The intuition here is that this operator largely acts like the original $\bowtie$, but that the transition system will keep any two sides that are joined by $\diamond$ well-separated at all times.

One way of achieving such behavior would be to use some kind of marking scheme. Whenever a new term of the form $(L \diamond R)$ is introduced, the occurrence of $\diamond$ is assigned a unique tag t. Then, the entire subterm L is "painted" with "color" t^- while R is painted with t^+. The idea is that colors can be mixed, but at no time can there be a mix involving both t^- and t^+ for any tag t. (Notice that distribution laws and β-reduction must propagate color information.)

While this seems to be a rather general approach because it can accommodate the possibility that certain occurrences of $\diamond$ be copied, colors will clutter our notation tremendously. Therefore, we use a much simpler (albeit less general) device: we make sure that if subterms M and N are separated by an occurrence of $\diamond$ before a transition, they must still be separated by a $\diamond$ after the transition.

It is easy to check that the rules for $\rightarrow_\diamond$ (see Figure 16) satisfy this requirement. In comparison with the original transition system for the $\lambda^{\bowtie}$-calculus (without constrained choice) there are some interesting aspects of the new rule set that are worthwhile pointing out:

- Constraining antecedents appear in the distribution rules for the *free* choice operator (as well as in the rules for β-reduction) because these rules would otherwise be able to make copies of $\diamond$.
- No constraining antecedents appear in distribution rules for constrained choice because even though they might make copies of occurrences of $\diamond$, the copies always end up on opposite sides of another $\diamond$.
- Associativity rules for mixed $\bowtie$ and $\diamond$ are not duals of one another. A constrained choice is permitted to pass a free choice when traveling outwards, thereby causing more subexpressions to be separated than before.[6] On the other hand, a free choice must turns into a constrained choice if it passes another constrained choice while traveling outwards (see also Figure 17).
- Whenever the presence of a $\diamond$ in some subterm inhibits one of the distribution rules for $\bowtie$, the rule can always

[6]This would not happen if we had used a coloring scheme as outlined earlier.

$$\frac{\diamond \text{ does not occur in } N}{(\lambda x.M\ N) \rightarrow_\diamond M[N/x]} \quad \text{(F-BETA)}$$

$$\frac{x \text{ appears free at most once in } M}{(\lambda x.M\ N) \rightarrow_\diamond M[N/x]} \quad \text{(C-BETA)}$$

$$\frac{N \rightarrow_\diamond N'}{M[N] \rightarrow_\diamond M[N']} \quad \text{(CONTEXT)}$$

$$(M \bowtie N) \rightarrow_\diamond (N \bowtie M) \quad \text{(F-SYMM)}$$

$$(M \diamond N) \rightarrow_\diamond (N \diamond M) \quad \text{(C-SYMM)}$$

$$(L \bowtie (M \bowtie N)) \rightarrow_\diamond ((L \bowtie M) \bowtie N) \quad \text{(FF-ASSOC)}$$

$$(L \bowtie (M \diamond N)) \rightarrow_\diamond ((L \bowtie M) \diamond N) \quad \text{(FC-ASSOC)}$$

$$(L \diamond (M \bowtie N)) \rightarrow_\diamond ((L \diamond M) \diamond N) \quad \text{(CF-ASSOC)}$$

$$(L \diamond (M \diamond N)) \rightarrow_\diamond ((L \diamond M) \diamond N) \quad \text{(CC-ASSOC)}$$

$$\lambda x.(M \bowtie N) \rightarrow_\diamond (\lambda x.M \bowtie \lambda x.N) \quad \text{(F-LAM)}$$

$$\lambda x.(M \diamond N) \rightarrow_\diamond (\lambda x.M \diamond \lambda x.N) \quad \text{(C-LAM)}$$

$$\frac{\diamond \text{ does not occur in } N}{((L \bowtie M)\ N) \rightarrow_\diamond ((L\ N) \bowtie (M\ N))} \quad \text{(F-APP-L)}$$

$$((L \diamond M)\ N) \rightarrow_\diamond ((L\ N) \diamond (M\ N)) \quad \text{(C-APP-L)}$$

$$\frac{\diamond \text{ does not occur in } L}{(L\ (M \bowtie N)) \rightarrow_\diamond ((L\ M) \bowtie (L\ N))} \quad \text{(F-APP-R)}$$

$$(L\ (M \diamond N)) \rightarrow_\diamond ((L\ M) \diamond (L\ N)) \quad \text{(C-APP-R)}$$

$$(M \bowtie N) \rightarrow_\diamond M \quad \text{(F-CHOOSE)}$$

$$(M \diamond N) \rightarrow_\diamond M \quad \text{(C-CHOOSE)}$$

Figure 16: Transition relation $\rightarrow_\diamond$ for the calculus with free choice $\bowtie$ and constrained choice $\diamond$. Notice how the rules maintain the invariant that subterms separated by $\diamond$ before a rule was applied will still be separated by $\diamond$ afterwards.

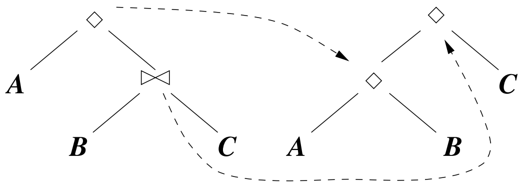

Figure 17: Rule CF-ASSOC in the calculus with free ($\bowtie$) and constrained ($\diamond$) choice. When travelling outwards, a $\bowtie$ turns into a $\diamond$ when passing another $\diamond$. Otherwise the subterms A and C which were on different sides of a $\bowtie$ before the transition would no longer be on different sides of any $\bowtie$ afterwards.

be re-enabled either by applying C-CHOOSE or by using distribution- and associativity laws until the offending $\diamond$ has been lifted out of the subterm.

5.2 When to use constrained choice

If one takes the code for structure `F` from above and translates it to the untyped lambda-calculus by simply stripping away all type- and module-information, one arrives at something like the following expression which represents `F` by building the pair of functions `f` and `a`:

$$\underbrace{(\texttt{pair}\ \overbrace{(\lambda xy.\cdots)}^{\texttt{F.f}}\ \overbrace{(\lambda fxy.(f\ y\ x))}^{\texttt{F.a}})}_{\text{structure } \texttt{F}}$$

On the other hand, the code for `F'` is:

$$\underbrace{(\texttt{pair}\ \overbrace{(\lambda yx.\cdots)}^{\texttt{F'.f}}\ \overbrace{\lambda f.f}^{\texttt{F'.a}})}_{\text{structure } \texttt{F'}}$$

These two expressions are not equivalent. Contexts that distinguish between the two are easily constructed. Therefore, rewriting the first into the second is not justified by the equational theory, and neither is rewriting it into a choice between the two.

The problem is that stripping away types has discarded essential information. Any context that would be able to distinguish between the two expressions is not well-typed according to ML's typing rules. *This* reasoning—as opposed to equational reasoning in the untyped calculus—is what enabled the compiler to legally rewrite `F` into `F'`.

In a second-order typed λ-calculus, abstract types such as the type `t` in structure `F` are explained in terms of existentially quantified type variables [12]. Thus, we can say that whenever one wants to track the choice between two expressions that have an existentially quantified type, one must use $\diamond$ instead of $\bowtie$. Examples for this exist in abundance and go beyond user-defined abstract types (although there is always some notion of abstraction at the root of the problem).

One abstraction that is built into every functional programming language is that of a *closure*. Closure representation is largely at the discretion of the compiler since it is clearly necessary to represent closures consistently. However, the compiler will eventually make the construction of closures explicit. If one is interested in tracking several possible representations, then the alternatives must again be kept separated by using $\diamond$ instead of $\bowtie$. This observation resonates well with the use of existential types for typed closure conversion as described by Minamide, Morrisett, and Harper [11].

5.3 Limitations of constrained choice

At the time constrained choices are decided, one of the two alternatives has to completely disappear from the program. This rules out any coexistence of two correct but incompatible implementations of the same abstraction within the same program.

In general, to prove that such coexistence would be harmless requires a dataflow analysis which must establish that the alternatives are always kept apart at runtime. (Of course, it would have been unreasonable to expect the simple addition of choice alone to solve such dataflow problems.)

5.4 Using wrappers

One way of eliminating the need for constrained choice is to adapt Leroy's wrapping-unwrapping technique for representation analysis [7], refinements of which have been studied by Shao [15, 16]. Recall the example of structure `F` with its abstract type `F.t` from the beginning of this section. We can retain knowledge of the abstract nature of `F.t` by keeping explicit conversions between concrete representations and abstract representations.

Assuming we have two ways of implementing `F.t`, we introduce conversion operation $\mathtt{wrap}_1(\cdot)$, $\mathtt{unwrap}_1(\cdot)$, $\mathtt{wrap}_2(\cdot)$, and $\mathtt{unwrap}_2(\cdot)$ from and to some "abstract" representation for each. It is then possible to translate structure `F` either into

$$(\mathtt{pair}\ (\mathtt{wrap}_1(\lambda xy.\cdots))\ (\lambda fxy.(\mathtt{unwrap}_1(f)\ y\ x)))$$

or into

$$(\mathtt{pair}\ (\mathtt{wrap}_2(\lambda yx.\cdots))\ \lambda f.\mathtt{unwrap}_2(f))$$

The trick is that the compiler "knows" how $\mathtt{wrap}_i(\cdot)$ cancels with a corresponding $\mathtt{unwrap}_i(\cdot)$:

$$\mathtt{wrap}_i(\mathtt{unwrap}_i(M)) \to M$$

$$\mathtt{unwrap}_i(\mathtt{wrap}_i(M)) \to M$$

Thus, an application of `F.a` to `F.f` is translated as $(\lambda fxy.(\mathtt{unwrap}_1(f)\ y\ x)\ \mathtt{wrap}_1(\lambda xy.\cdots))$ which then turns into $\lambda xy.(\mathtt{unwrap}_1(\mathtt{wrap}_1(\lambda xy.\cdots))\ y\ x)$ at which point the wrapper cancels with the unwrapper so that the final result is $\lambda xy.(\lambda xy.\cdots yx)$. Similarly, had we used `F'`, then $\mathtt{wrap}_2(\cdot)$ would have cancelled with $\mathtt{unwrap}_2(\cdot)$, the final result being the efficient version of `F'.f`: $(\lambda yx.\cdots)$.

The point of all this is that we can now join the two versions using *free* choice because the wrappers will sufficiently shield one concrete version of the abstract type from the other. The worst that could happen is that the final program still contains wrapper-unwrapper pairs of the form $\mathtt{wrap}_1(\mathtt{unwrap}_2(\cdot))$ or some permutation thereof.

Of course, on the way to an executable form of the program one must give a concrete meaning—an implementation—for such residual wrap-unwrap code. To do this, we single out one representation as the *canonical* representation. Operation $\mathtt{wrap}_i(M)$ implements the transformation of some expression M_i that currently uses representation i to a corresponding canonical $\hat{M}$; $\mathtt{unwrap}_i(\hat{M})$ is the inverse. In our example, if variant 1 is singled out as "canonical," then $\mathtt{wrap}_1(\cdot)$ and $\mathtt{unwrap}_1(\cdot)$ are identity transformations while for $\mathtt{wrap}_2(\cdot)$ and $\mathtt{unwrap}_2(\cdot)$ one could give the following transition rules:

$$\mathtt{wrap}_2(M) \to \lambda xy.(M\ y\ x)$$

$$\mathtt{unwrap}_2(M) \to \lambda xy.(M\ y\ x)$$

(Notice that these rules can be used to derive the cancellation rules that we have seen earlier.)

The one drawback of this approach is that it can lead to time- and space leaks. It is not difficult to construct a loop where the same value repeatedly goes through wrapping- and unwrapping-transformations, each such transformation constructing a new closure around the previous one. As a result, closures "pile up", programs begin to leak space and can suffer dramatic changes in their asymptotic time complexity. (The problem is that unwrapping is not *really* the inverse of wrapping in terms of what happens at low level.)

Minamide and Garrigue have demonstrated a method of wrapping and unwrapping that does not suffer from this problem [10]. Instead of passing a value in one *or* the other representation they pass it as a pair of *both* representations. Monnier pointed out that this could be seen as a way of representing a *constrained choice* between the two versions at runtime—together with a compile-time guarantee that in each situation the respective correct alternative will be selected [14].

6 Heuristic measuring

As we have already discussed, a compiler's optimization decisions are driven by heuristics. These heuristics typically use some kind of *measuring function* (or just "measures") to either assess overall code quality or to estimate in advance what the effect of a transformation will be. Having choice operators in the intermediate language does not eliminate the need for either the heuristics or the measuring that goes along with them.

We will now try to convince ourselves that it is (typically) possible to extend existing measures for the choice-free core to the full $\lambda^{\bowtie}$-calculus without degrading their predictive powers.

6.1 Compositional monotone measures

For any $M \in E_\lambda$, let $\mathcal{M}[\![M]\!] \in \mathbb{N}$ be some number "measuring" M so that the compiler will consider "M better than N" whenever $\mathcal{M}[\![M]\!] < \mathcal{M}[\![N]\!]$.

To extend $\mathcal{M}[\![\cdot]\!]$ to expressions in $E_{\bowtie}$ (or $E_\diamond$) we note that $\mathcal{M}[\![\cdot]\!]$ already gives us a way of measuring each commit $M' \in \mathcal{C}[\![M]\!]$ for a given $M \in E_{\bowtie}$. Thus, we define:

$$\mathcal{M}[\![M]\!] \ \ \underset{def}{=} \ \ \min_{M' \in \mathcal{C}[\![M]\!]} \mathcal{M}[\![M']\!]$$

This definition reflects our intention of eventually selecting the "best" candidate from the commit set. However, it does not provide an efficient way of calculating $\mathcal{M}[\![M]\!]$ because the size of the commit set is exponential in the number of choices within M. (Of course, an upper bound on $\mathcal{M}[\![M]\!]$ can be obtained simply by taking an arbitrary element in $\mathcal{C}[\![M]\!]$ and measuring that. But the so-calculated upper bound could be very far from the actual value and result in an optimizer that acts "paranoid.")

Fortunately, measuring functions in real compilers are often *compositional* and *monotone*:

compositional: The measure is defined in a "syntax-directed" way, i.e., there is a set of rules that describes $\mathcal{M}[\![M]\!]$ for any M purely in terms of $\mathcal{M}[\![N_1]\!], \ldots, \mathcal{M}[\![N_k]\!]$ where $N_1, \ldots, N_k$ are non-overlapping subterms of M. Moreover, for each concrete M there is precisely one rule that applies.

monotone: In the set of rules defining $\mathcal{M}[\![M]\!]$, the functions that map $\mathcal{M}[\![N_1]\!], \ldots, \mathcal{M}[\![N_k]\!]$ to $\mathcal{M}[\![M]\!]$ are monotone in each argument.

Perhaps *the* most popular kind of measure for traditional (non-choice-based) intermediate languages is an estimate of code size. Appel [1, Chapter 7.2] describes in quite some detail what the size measure $\mathcal{O}[\![\cdot]\!]$ for SML/NJ's CPS language is and how it can be used in making optimization decisions. It is easy to see that Appel's definition of $\mathcal{O}[\![\cdot]\!]$ is both compositional and monotone.

6.2 Measuring $M \in E_{\bowtie}$ efficiently

To make the example concrete, let us define our own (somewhat arbitrary) size measure $\mathcal{O}_\lambda[\![\cdot]\!]$ for E_λ as follows:

$$\begin{array}{rcl}\mathcal{O}_\lambda[\![x]\!] & = & c_1 \\ \mathcal{O}_\lambda[\![\lambda x.M]\!] & = & c_2 + \mathcal{O}_\lambda[\![M]\!] \\ \mathcal{O}_\lambda[\![(M\ N)]\!] & = & c_3 + \mathcal{O}_\lambda[\![M]\!] + \mathcal{O}_\lambda[\![N]\!]\end{array}$$

(Here, c_1, c_2, and c_3 are small constants.) Obviously, $\mathcal{O}_\lambda[\![\cdot]\!]$ is compositional and monotone.

With this we now define a new measure $\mathcal{O}_{\bowtie}[\![\cdot]\!]$ for $\lambda^{\bowtie}$-expressions:

$$\begin{array}{rcl}\mathcal{O}_{\bowtie}[\![x]\!] & = & c_1 \\ \mathcal{O}_{\bowtie}[\![\lambda x.M]\!] & = & c_2 + \mathcal{O}_{\bowtie}[\![M]\!] \\ \mathcal{O}_{\bowtie}[\![(M\ N)]\!] & = & c_3 + \mathcal{O}_{\bowtie}[\![M]\!] + \mathcal{O}_{\bowtie}[\![N]\!] \\ \mathcal{O}_{\bowtie}[\![(M \bowtie N)]\!] & = & \min\{\mathcal{O}_{\bowtie}[\![M]\!], \mathcal{O}_{\bowtie}[\![N]\!]\}\end{array}$$

Based on the monotonicity of $\mathcal{O}_\lambda[\![\cdot]\!]$, a simple inductive argument shows that, indeed,

$$\mathcal{O}_{\bowtie}[\![M]\!] = \min_{M' \in \mathcal{C}[\![M]\!]} \mathcal{O}_\lambda[\![M']\!]$$

.

Moreover, $\mathcal{O}_{\bowtie}[\![M]\!]$ can be efficiently calculated in time linear to the size of M. For $M \in E_\lambda$ it coincides with $\mathcal{O}_\lambda[\![M]\!]$. And finally, calculation of the new measure provides an effective way of picking the right element from the commit set, i.e, the $M_{\text{best}} \in \mathcal{C}[\![M]\!]$ for which $\mathcal{O}_{\bowtie}[\![M]\!] = \mathcal{O}_\lambda[\![M_{\text{best}}]\!]$.

In the same style, it is then possible to extend any compositional monotone measure $\mathcal{M}[\![\cdot]\!] : E_\lambda \to \mathbb{N}$ to arbitrary $\lambda^{\bowtie}$-expressions in such a way that it can be efficiently calculated. Thus, for the rest of the discussion we will assume all measures to be of this kind.

6.3 Estimating savings

The intuition behind the "minimum rule" when measuring choice can also be explained in terms of how the measure is used during optimization. In a situation where the compiler has a two or more alternative ways of applying $\rightarrow_\lambda$-transitions to a $\lambda^{\bowtie}$-term M, it can always first go to $M_{\text{best}} \in \mathcal{C}[\![M]\!]$, decide (the old-fashioned way based on $\mathcal{M}[\![M_{\text{best}}]\!]$) how it would proceed there, and then translate the decision back to M. The minimum rule provides this behavior without actually having to take the detour through M_{best}.

Thus, the "ballast" carried around in pending choices cannot make things worse. It may currently *look* worse than M_{best}, but if it really turns out to *be* worse, then it can always be dropped later. If, on the other hand, it enables some later optimization that $\mathcal{M}[\![M]\!]$ was not able to foresee, then all the better.

Thus, one simple way of estimating the savings of a transformation in the $\lambda^{\bowtie}$-calculus is to essentially use the same mechanism as the one used in the λ-calculus.

6.4 Pruning

Applying Choose corresponds to pruning the optimization search space. Since many transformations duplicate subterms, pruning must be used aggressively to keep the total size of a program's representation manageable.

In the calculus without choice, the same size measure $\mathcal{O}_\lambda[\![\cdot]\!]$ can be used both for estimating the size of the machine code generated for the program and for calculating the size of its current representation (because the two are closely related). With choice, this is no longer the case since the current representation also includes all the choices that will not make it into the final program, but $\mathcal{O}_{\bowtie}[\![\cdot]\!]$ does not measure those.

A function $\mathcal{R}_{\bowtie}[\![\cdot]\!]$ measuring representation size instead of expected code size can be defined simply by using summation to handle choice:

$$\begin{array}{rcl}\mathcal{R}_{\bowtie}[\![x]\!] & = & c_1 \\ \mathcal{R}_{\bowtie}[\![\lambda x.M]\!] & = & c_2 + \mathcal{R}_{\bowtie}[\![M]\!] \\ \mathcal{R}_{\bowtie}[\![(M\ N)]\!] & = & c_3 + \mathcal{R}_{\bowtie}[\![M]\!] + \mathcal{R}_{\bowtie}[\![N]\!] \\ \mathcal{R}_{\bowtie}[\![(M \bowtie N)]\!] & = & \mathcal{R}_{\bowtie}[\![M]\!] + \mathcal{R}_{\bowtie}[\![N]\!]\end{array}$$

$\mathcal{R}_{\bowtie}[\![\cdot]\!]$ helps detecting when a term M "needs pruning", but we use $\mathcal{O}_{\bowtie}[\![\cdot]\!]$ (or whatever other measure $\mathcal{M}[\![\cdot]\!]$ is appropriate) to decide where within M to actually do the pruning. For example, one could search M to find an occurrence of $(N \bowtie N')$ that maximizes $|\mathcal{O}_{\bowtie}[\![N]\!] - \mathcal{O}_{\bowtie}[\![N']\!]|$ and then drop the worse alternative.

An algorithm for this approach would run in linear time, but it might not do exactly the right thing since locally maximizing the difference of the term to be thrown away and its alternative does not necessarily have the same effect globally. From the global point of view, any pruning replaces $M = L[(N \bowtie N')]$ with $M' = L[N]$ for some context L. We want to pick L (and therefore N and N') in such a way that $\mathcal{O}_{\bowtie}[\![L[N']]\!]$ is maximized. This leads to a simple quadratic-time solution. Of course, pruning might have to be repeated because a single incision will not always bring $\mathcal{R}_{\bowtie}[\![M]\!]$ back within the limit.

For certain measure functions one can improve on the quadratic-time search algorithm by taking advantage of regularities. In particular, if the compositional rules that define $\mathcal{M}[\![\cdot]\!]$ relate the measures of M's subterms to that of M in an arithmetically linear fashion, then the search for the globally best choice to be pruned can be done in linear time. Notice that our size measure $\mathcal{O}_\lambda[\![\cdot]\!]$ has this linearity property, and so does Appel's size measure for CPS-terms. In fact, all the factors are 1, so even the above-mentioned search for the locally best pruning would work.

6.5 Managing non-termination

One problem with the $\text{Beta}_{\bowtie}$ rule is that after it has been applied, the original β-redex is still there. Therefore, if one is not careful, this could be a source of non-termination in the optimizer.

Of course, there is no shortage of potential non-termination even with the original choice-free calculus, and optimizing compilers had to deal with this before. Most of the time this is simply a consequence of the fact that an optimizer can never be "perfect" because it cannot beat the halting problem.

In the particular case of $\text{BETA}_{\bowtie}$, however, turning $(\lambda x.M\,N)$ first into $(\lambda y.M[(y \bowtie N)/x]\,N)$ and then immediately into $(\lambda z.M[(y \bowtie N)/x][(z \bowtie N)/y]\,N)$ is not just something that can lead to non-termination, it also is completely useless because it never provides any progess towards the goal of optimization! Since all it does is to build choice-triplets of the form $((z \bowtie N) \bowtie N)$, there is no extra information being propagated. Therefore, the second $\text{BETA}_{\bowtie}$ step does not help the optimizer in any way at this time.

A simple solution is to mark β-redexes that have been $\text{BETA}_{\bowtie}$-reduced so they will not be eligible for future applications of this rule. On the other hand, since it is possible that pruning prematurely undoes the effect of $\text{BETA}_{\bowtie}$, it might be a better approach to adjust the heuristics in such a way that while they make re-application of $\text{BETA}_{\bowtie}$ extremely unlikely, they do not prohibit it altogether.

7 Conclusions and future work

We have investigated a simple addition to λ-calculus. The choice operator $\bowtie$ enables compilers to encode pending optimization decisions into terms of the calculus itself and to perform breadth-first searches in optimization space. While born out of the work on cross-module inlining, this can be generally beneficial for intermodule and intra-module optimizers.

Because of the introduction of a nondeterministic operator, the resulting calculus loses confluence. However, we introduced the notion of non-ambiguity, showed that it is preserved by the transition relation of the calculus, and that it provides a sufficiently strong notion of confluence for the segment of the system that a compiler would actually be interested in.

The presence of choice in the calculus does not eliminate the need for heuristics. While it will certainly be possible to design new heuristics that specifically take advantage of choice, we have argued here that adaptation of existing heuristics for choice-free calculi (as far as they are based on compositional monotone measures) is possible and will yield results that are no worse than before. In the future, an actual practical advantage of using intermediate languages with choice has to be verified by providing an implementation in a real compiler.

References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[3] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, volume 29, pages 13–23. ACM Press, June 1994.

[4] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 112–124. ACM Press, June 1997.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, New York, 1993. ACM Press.

[6] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[7] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, January 1992. ACM Press.

[8] Xavier Leroy. The Objective Caml system (release 2.04). (with Didier Rémy, Jérôme Vouillon and Damien Doligez), Institut National de Recherche en Informatique et en Automatique, November 1999.

[9] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[10] Yasuhiko Minamide and Jacques Garrigue. On the run-time complexity of type-directed unboxing. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 1–12. ACM Press, September 1998.

[11] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, January 1996.

[12] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, July 1988.

[13] Eugenio Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, pages 14–22. IEEE, 1989.

[14] Stefan Monnier. personal communication, January 2000.

[15] Zhong Shao. *Compiling Standard ML for Efficient Execution on Modern Machines.* PhD thesis, Princeton University, Princeton, NJ, November 1994. Tech Report CS-TR-475-94.

[16] Zhong Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 85–98, New York, 1997. ACM Press.

[17] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, June 1997.

[18] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *1996 SIGPLAN Conference on Programming Language Design and Implementation*, New York, 1996. ACM Press.