λ to SKI, Semantically
Declarative Pearl

Oleg Kiselyov(✉)

Tohoku University, Sendai, Japan
oleg@okmij.org

Abstract. We present a technique for compiling lambda-calculus expressions into SKI combinators. Unlike the well-known bracket abstraction based on (syntactic) term re-writing, our algorithm relies on a specially chosen, *compositional* semantic model of generally open lambda terms. The meaning of a closed lambda term is the corresponding SKI combination. For simply-typed as well as unityped terms, the meaning derivation mirrors the typing derivation. One may also view the algorithm as an algebra, or a non-standard evaluator for lambda-terms (i.e., denotational semantics).

The algorithm is implemented as a tagless-final compiler for (uni)typed lambda-calculus embedded as a DSL into OCaml. Its type preservation is clear even to OCaml. The correctness of both the algorithm and of its implementation becomes clear.

Our algorithm is easily amenable to optimizations. In particular, its output and the running time can both be made linear in the size (i.e., the number of all constructors) of the input De Bruijn-indexed term.

1 Introduction

Since Curry [2] definitely and constructively demonstrated that any lambda-expression can be transformed into SKI-combinators, there seems to have been no need to revisit this issue. And yet it has continued to attract attention: of mathematicians, investigating connections of lambda-terms and graphs, and of theoretical computer scientists, studying the complexity of this process [4].

In a surprising turn, the translation from lambda-terms to SKI combinators, previously regarded as purely academic, proved very practical. David Turner used the translation as the compilation technique for his functional language SASL [12,13], and later Miranda. The familiar presentation of the translation, or compilation – called “the bracket abstraction” and originally due to Schoenfinkel [8] – was made popular by Turner, who polished and optimized it, and made it practical. The entire chapter of Peyton Jones’ book on implementing functional languages [7, Chap. 16] is devoted to the SKI translation, which is “appealing because it gives rise to an extremely simple reduction machine”. The book mentions two physical machines designed around SKI reductions: Cambridge SKIM machine [11] and Burroughs NORMA.

© Springer International Publishing AG, part of Springer Nature 2018
J. P. Gallagher and M. Sulzmann (Eds.): FLOPS 2018, LNCS 10818, pp. 33–50, 2018.
https://doi.org/10.1007/978-3-319-90686-7_3

Above all, the bracket abstraction is a pearl. We can't help to peek at its shine right away, ahead of the formal presentation in the background Sect. 2. The translation turns a lambda-term to applications of the S, K, I combinators, whose reductions are shown in the left column of Fig. 1.

	Reductions	Compilation Rules
I	$Ix \rightsquigarrow x$	$\lambda x.\, x \mapsto I$
K	$K\,y\,x \rightsquigarrow y$	$\lambda x.\, e \mapsto Ke$ †
S	$S f g\, x \rightsquigarrow fx(gx)$	$\lambda x.\, e_1\; e_2 \mapsto S\; (\lambda x.\, e_1)\; (\lambda x.\, e_2)$

Fig. 1. SKI reductions and compilation rules. †In the K compilation rule, e is a combinator or variable other than x.

The translation is re-writing with three simple rules of the right column of Fig. 1. The I and K rules are the re-statements of the corresponding reductions; the S rule becomes obvious if we notice that a term e with a possibly free variable x is equal to $(\lambda x.\, e)x$. Taking as the running example $\lambda x.\, \lambda y.\, y\; x$, we can only use the S-rule, on the inner lambda-abstraction, obtaining $\lambda x.\, S\; (\lambda y.\, y)\; (\lambda y.\, x)$, to which K and I rules apply, giving $\lambda x.\, (SI)\; (Kx)$. Using the S rule three more times leads eventually to $S(S(KS)(KI))(S(KK)I)$. The result is bigger than the original term: we tackle the size explosion in Sect. 6.

We present another pearl of the translation from lambda-terms to SKI combinators and show off its facets. It comes from a very different oyster. Our translation is *not* based on (syntactic, in its essence) re-writing. Rather, we define a semantic model of (generally open) lambda-terms in terms of combinators, along with the way to compositionally compute the meaning of a term in that semantics from the meanings of its immediate children. The meaning of a closed lambda-term is designed to be the corresponding SKI term. Our translation stands out in avoiding operations like checking variable equality or free occurrences. Whereas the bracket abstraction cannot do anything meaningful with the mere x or $x\; y$ subterms, ours can. As a source we use lambda-terms with De Bruijn indices; it turns out the indices supply just enough information about the environment to figure out the meaning of a single variable or a combination of variables.

All in all, the semantic presentation of the SKI compilation avoids the 'nominal trench' of lambda-calculus; it is easier to see correct, easier to generalize and optimize. The semantic pearl shines brighter.

The Highlights

– Section 3 develops the semantic-based translation intuitively and formally. We start with the simply-typed calculus to see the correspondence of type and meaning derivations. Already the simplest polish naturally reveals optimizations.

- The semantic translation is straightforward to realize in tagless-final style: Sect. 4. The OCaml implementation highlights the algebra of the translation, naturally prompting further optimizations.
- Section 5 extends the calculus with integers, conditional and general recursion. The translation becomes practical.
- Section 6 presents the linear space and time translation, for the general untyped calculus (which applies to the typed calculus as well). It goes beyond the Schoenfinkel, Curry and Turner bracket abstraction and their further optimizations such as director strings; Sect. 7 discusses how much beyond and in which direction. Our translation is hence the viable alternative to supercombinators.

We start with the background, in the next section. The complete OCaml code is available at http://okmij.org/ftp/tagless-final/skconv.ml.

2 Lambda- and SKI-calculi and the Bracket Abstraction

This background section recapitulates lambda- and combinator calculi and the classical bracket abstraction. Mainly, it introduces the notation for the rest of the paper.

Figure 2 presents the syntax of the calculi and the notational conventions, heavily used throughout. We write e for expressions in lambda-calculus with names and e^0 for expressions in the calculus with De Bruijn indices – although we often write just e if the context disambiguates. Lower-case f, g, x, y, u, v (possibly adorned with subscripts or superscripts) are always variables. We consider both untyped calculi and simply-typed calculi. In the latter case, types (denoted by $\alpha, \beta, \gamma, \sigma, \tau$ metavariables) are base and arrow types; there are no type variables. Γ denotes a possibly empty sequence of types, whereas Γ^+ stands for a nonempty sequence. We write τ, Γ and Γ, τ for prepending, resp. appending τ to the sequence Γ, and Γ_1, Γ_2 for sequence concatenation. The S, K, I and other combinators are, by convention, upper-case letters; the metavariable d stands for an arbitrary combinator expression.

The meaning of combinators is defined by their reduction rules, collected in Fig. 3. It presents not only S, K, and I but also B and C combinators, which we

Variables	f, g, x, y, u, v
Base Types	ι
Types	$\alpha, \beta, \gamma, \sigma, \tau ::= \iota \mid \tau \rightarrow \tau$
Type Environment	$\Gamma ::= \tau, \ldots$
Expressions	$e ::= x \mid \lambda x.\, e \mid e\ e$
De Bruijn Expressions	$e^0 ::= z \mid s\ e^0 \mid \lambda\, e^0 \mid e^0\ e^0$
Combinator Expressions	$d ::= d\ d \mid S \mid K \mid I \mid B \mid C$

Fig. 2. Syntax of languages

$$\begin{array}{lll} I\ x & \rightsquigarrow x \\ K\ y\ x & \rightsquigarrow y \\ S\ f\ g\ x & \rightsquigarrow f x\,(g x) \end{array} \qquad \begin{array}{l} B\ f\ g\ x \rightsquigarrow f\,(g x) \\ C\ f\ g\ x \rightsquigarrow f\,x\,g \end{array}$$

Fig. 3. Combinators and their reduction rules

shall use later. They are particular cases of S; B is the functional composition. All combinators can be expressed in terms of just S and K.

Figure 4, borrowed from [7, Fig. 16.2] with the adjusted notation, presents the basic bracket abstraction algorithm: the formalization of the procedure intuitively described in Sect. 1. The main translation $\mathcal{C}\,[e]$ uses the auxiliary $\mathcal{A}_x\,[e']$ to "abstract" x from a lambda-free expression e' and produce the corresponding combinator expression. As shown in Sect. 1, $\mathcal{C}\,[\lambda x.\,\lambda y.\,y\ x]$ gives $S(S(KS)(KI))(S(KK)I)$.

$\mathcal{C}\,[e]$: compile e to combinators

$$\begin{array}{ll} \mathcal{C}\,[e_1\ e_2] & \mapsto \mathcal{C}\,[e_1]\ \ \mathcal{C}\,[e_2] \\ \mathcal{C}\,[\lambda x.\,e] & \mapsto \mathcal{A}_x\,[\mathcal{C}\,[e]] \\ \mathcal{C}\,[c] & \mapsto c \end{array}$$

$\mathcal{A}_x\,[e]$: abstract x from e

$$\begin{array}{ll} \mathcal{A}_x\,[e_1\ e_2] & \mapsto S\ (\mathcal{A}_x\,[e_1])\ (\mathcal{A}_x\,[e_2]) \\ \mathcal{A}_x\,[x] & \mapsto I \\ \mathcal{A}_x\,[c] & \mapsto K\ c \quad \text{where } c \text{ is not } x \end{array}$$

Fig. 4. Basic Bracket Abstraction. We write c for a variable, constant or a combinator expression. $\mathcal{A}_x\,[e]$ applies to e with no inner lambdas.

As another example, $\lambda y.\,(\lambda x.\,x\,x)(\lambda x.\,x\,x)$ is translated (in a less naive way) to $K((SII)(SII))$. The lambda-term is not strongly normalizing (i.e., has a divergent reduction sequence) and the same holds for the translated SKI term. With no reductions under lambda (as in call-by-name or call-by-value), the lambda-term is in normal form. In SKI this corresponds to the head reduction strategy: $d\,d_1\,\ldots\,d_n$, where d is a combinator, is irreducible if no reduction rule applies to d. In the following, only head reductions are considered.

3 Semantic Translation

This section formally presents our translation, first intuitively and then formally. We argue about its correctness in Sect. 4 and improving memory and run-time performance in Sect. 6. The translation is formulated as a non-standard denotation of lambda-terms and amounts to a constructive proof of the combinatorial completeness of lambda-calculus.

Although our translation applies both to typed and untyped calculi, the intuitions are easier to see with types, such as those in Fig. 5. (We explicitly consider the untyped case in Sect. 6.)

This is the very standard type system for simply-typed lambda-calculus, conventionally presented as the inference rules for the judgment $\Gamma \vdash e : \tau$ that

$$\frac{}{\tau \vdash z : \tau}\,Var \qquad \frac{\Gamma \vdash e : \tau}{\sigma, \Gamma \vdash e : \tau}\,WL \qquad \frac{\Gamma^+ \vdash e : \tau}{\Gamma^+, \sigma \vdash s\ e : \tau}\,WR$$

$$\frac{\Gamma, \sigma \vdash e : \tau}{\Gamma \vdash \lambda\, e : \sigma \to \tau}\,Abs \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}\,App$$

Fig. 5. The simple type system of the De Bruijn lambda-calculus

a term e has the type τ in the environment Γ. The latter is a *sequence* of types. Figure 5 is more explicit than usual about structural rules, distinguishing the right weakening[1] (WR), which is syntactically marked as $s\ e$, from the left weakening (WL), which is unmarked. The 'unhinged' nature of (WL) prompts us to move to a different type system, in which every rule is connected to some syntactic feature and derivations are syntax-directed. The left-hand-side of Fig. 6 describes such a system – to be called 'leftless', in contrast with the 'lefty' system of Fig. 5, from which it was derived by working the (WL) rule into the others. The new type system is equivalent to the old, as the following proposition shows.

Definition 1. *A type judgment $\Gamma \vdash e : \tau$ (and its derivation) are called* left-strong *if (i) Γ is empty, or (ii) Γ is σ, Γ' and $\Gamma' \vdash e : \tau$ is not derivable.*

Proposition 1. *Any left-strong judgment (derivation) in the lefty system can be derived (converted to) the leftless system, and vice versa.*

In the forward direction, the proof is by the straightforward induction on the type derivation. The reverse direction is trivial.

The reason to split hairs about the weakening and to introduce the messier leftless system is to make clearer the correspondence between type and semantic derivations.

We now introduce our denotational semantics. Generally, denotational semantics assigns each syntactic object an element of some semantic domain, which serves as the 'meaning' for the object. The assignment must be compositional: the meaning of an object should depend only on the meanings of its immediate subcomponents. In Church-style calculi, only well-typed terms 'make sense'. Therefore, the meaning is assigned to type derivations, represented by the judgement in their conclusion: in symbols, $\mathcal{E}\,[\vdash e : \tau] \in \mathcal{V}\,[\tau]$ where $\mathcal{V}\,[-]$ stands for the semantic domain, also type-indexed. To give the meaning to an open term, however, we need to know what its free variables mean. The common approach is to take the term denotation to be a function of an 'environment', which maps each free variable to its meaning. When variables are represented by De Bruijn indices, the environment may be realized as a tuple: $\mathcal{E}\,[\tau_n, \ldots, \tau_1 \vdash e : \tau] \in \mathcal{V}\,[\tau_n \times \ldots \times \tau_1 \to \tau]$.

[1] By 'weakening' we mean a (structural) inference rule stating that adding more premises to hypotheses of a valid logical deduction preserves the validity.

$$\begin{array}{cc}
\Gamma \vdash e : \tau & \mathcal{E}\,[\Gamma \vdash e : \tau] \\
\hline
\dfrac{}{\tau \vdash z : \tau}\,Var & \dfrac{}{\tau \models I}\,EVar \\
\dfrac{\Gamma^+ \vdash e : \tau}{\Gamma^+, \sigma \vdash s\ e : \tau}\,W & \dfrac{\Gamma^+ \models d}{\Gamma^+, \sigma \models (\models K)\,\text{II}\,(\Gamma^+ \models d)}\,EW \\
\dfrac{\vdash e : \tau}{\vdash \lambda e : \sigma \to \tau}\,Abs_0 & \dfrac{\models d}{\models K\, d}\,EAbs_0 \\
\dfrac{\Gamma, \sigma \vdash e : \tau}{\Gamma \vdash \lambda e : \sigma \to \tau}\,Abs & \dfrac{\Gamma, \sigma \models d}{\Gamma \models d}\,EAbs \\
\dfrac{\Gamma_1 \vdash e_1 : \sigma \to \tau \quad \Gamma_2 \vdash e_2 : \sigma}{\Gamma_1 \sqcup \Gamma_2 \vdash e_1\ e_2 : \tau}\,App & \dfrac{\Gamma_1 \models d_1 \quad \Gamma_2 \models d_2}{\Gamma_1 \sqcup \Gamma_2 \models (\Gamma_1 \models d_1)\,\text{II}\,(\Gamma_2 \models d_2)}\,EApp
\end{array}$$

$$\text{where}\quad \begin{aligned} \Gamma_1 \sqcup \Gamma_2 &= \Gamma_1 \text{ if } \Gamma_1 = \Gamma_3, \Gamma_2 \text{ for some } \Gamma_3 \\ &= \Gamma_2 \text{ if } \Gamma_2 = \Gamma_3, \Gamma_1 \end{aligned}$$

Fig. 6. The 'leftless' type system and the corresponding denotational semantics: the rules for deriving $\Gamma \vdash e : \tau$ (left column) and $\mathcal{E}\,[\Gamma \vdash e : \tau]$ (right column). The function $\Gamma_1 \sqcup \Gamma_2$ checks that one sequence is a suffix of the other and returns the longer one. The semantic function II is described in text.

Now come two key ideas. Without special introduction they are easy to miss due to their simplicity. First, we curry the denotation: $\mathcal{E}\,[\tau_n, \ldots, \tau_1 \vdash e : \tau] \in \mathcal{V}\,[\tau_n \to \ldots \to \tau_1 \to \tau]$. Second, as the semantic domain $\mathcal{V}\,[\tau]$ we take the set of combinator expressions of type τ. (We shall soon see it is non-empty.) Alas, we have conflated closed and open terms: the combinator $I : \tau \to \tau$ may denote the closed term $\vdash \lambda z : \tau \to \tau$ as well as the open term $\tau \vdash z : \tau$. To distinguish the denotations of terms with the different number of free variables, we pair the combinator expression with Γ[2].

Definition 2. *The denotation of a typed lambda-term $\mathcal{E}\,[\Gamma \vdash e : \tau]$ where Γ is $\tau_n, \ldots, \tau_1$ is a tuple of an SKI term d of the type $\tau_n \to \ldots \to \tau_1 \to \tau$, and the type sequence Γ. We write such tuple as $\Gamma \models d$.*

It follows that for a closed e, $\mathcal{E}\,[\vdash e : \tau]$ is a combinator expression of the type τ, which we take to be the result of our SKI translation[3]. The denotation $\mathcal{E}\,[\tau \vdash z : \tau]$ of the open term z is clearly $\tau \models I$: indeed, I, when applied to some d_0, the one-component 'environment', reduces to that d_0 – the behavior

[2] As we will see in Sect. 6, it is enough to keep the length of Γ, that is, the number of free variables in a term.

[3] It is natural to wish a denotation of an open term be non-divergent: if $\tau_n, \ldots, \tau_1 \models d$ then d, until applied to n other terms, should have only a finite number of reductions, if any at all. The wish is already granted: in the present simply-typed calculus, all terms are (strongly) normalizing. We have to wait until Sect. 6 to say something non-trivial about termination.

expected of z. Likewise, we find that $\mathcal{E}\,[\tau, \sigma \vdash s\, z : \tau]$ is $\tau, \sigma \models K$: the combinator K, applied to d_1 and d_0, the two-component environment, reduces to d_1.

The right-hand side column of Fig. 6 describes the compositional computation of $\mathcal{E}\,[\Gamma \vdash e : \tau]$ in the form of inference rules. (EVar) was explained already. The (EAbs$_0$) rule says that if the function's body is closed it is the constant function. (EAbs) amounts to η-conversion: if d is such that $(\tau_n, \ldots, \tau_1, \sigma \models d)$, then $(d\ d_n \ldots\ d_1)$ acts as a function: when applied to an argument $d_0 : \sigma$, that is, $d\ d_n \ldots\ d_1\ d_0$, it looks like d in the environment extended with d_0. The (EW) rule states that weakening is the application of the K combinator.

The semantic function $(\Gamma' \models d') \amalg (\Gamma \models d)$ computes the application: that is, converts $(\tau_n, \ldots, \tau_1 \models d')(\tau_m, \ldots, \tau_1 \models d)$ to the form $\tau_{\max\, n\, m}, \ldots, \tau_1 \models \bar{d}$ for some combinator expression $\bar{d}$. It is an exercise in combinatory logic. Its simplest (albeit not optimal) solution is to define $\amalg$ by induction[4]: Let's consider the case of the application of a closed term: $(\models d')\ \amalg\ (\tau_n, \ldots, \tau_1 \models d)$. Our goal is to represent $d'\,(d\ d_n \ldots d_1)$ as a combinator expression $\bar{d}$ applied to d_n through d_1. In the base case of $n = 0$, clearly $\bar{d}$ is $d'\ d$. In the inductive case, the B reduction rule from Fig. 3 gives us $d'\ (d\, d_n \ldots d_1) = (Bd')\,(d\, d_n \ldots d_2)\, d_1$. Therefore, $\bar{d}$ is the solution to the smaller instance of the problem: representing $(Bd')\,(d\ d_n \ldots d_2)$ as $\bar{d}\ d_n \ldots d_2$. In the general case of $(d'\ d_n \ldots d_1)(d\ d_m \ldots d_1)$ with $n \geq 1, m \geq 1$, the S reduction rule gives us $(S\,(d'\ d_n \ldots d_2))\ (d\ d_m \ldots d_2)\ d_1$. Then we look for $\bar{d'}$ such that $S\,(d'\ d_n \ldots d_2) = \bar{d'}\ d_n \ldots d_2$ (the earlier case of the closed-term application) and, finally, solve $(\bar{d'}\ d_n \ldots d_2)(d\ d_m \ldots d_2)$, which is the shorter version of the original problem. All in all, we obtain the following structurally recursive definition:

$$\begin{aligned}
(\models d')\ \amalg\ (\models d) &= d'\ d \\
(\models d')\ \amalg\ (\tau_n, \ldots, \tau_1 \models d) &= (\models Bd')\ \amalg\ (\tau_n, \ldots, \tau_2 \models d) \\
(\tau_n, \ldots, \tau_1 \models d')\ \amalg\ (\models d) &= (\models CCd)\ \amalg\ (\tau_n, \ldots, \tau_2 \models d') \\
(\tau_n, \ldots, \tau_1 \models d')\ \amalg\ (\tau_m, \ldots, \tau_1 \models d) &= (\tau_n, \ldots, \tau_2 \models (\models S)\ \amalg\ (\tau_n, \ldots, \tau_2 \models d'))\amalg(\tau_m, \ldots, \tau_2 \models d)
\end{aligned}$$

Figure 7 shows the typing and semantic derivations for the running example: the flipped application $\lambda x.\, \lambda y.\, y\ x$. The typing derivation can be read as a proof, from the (Var) axioms down to the conclusion that the sample term has the type $\alpha \to (\alpha \to \beta) \to \beta$. Likewise, the semantic derivation produces the meaning of the term just as compositionally, from the (EVar) axioms down the chain of inference rules. The less-obvious step is the $\amalg$ computation in the (EApp) rule:

$$\begin{aligned}
&(\alpha \to \beta \models I)\,\amalg\,(\alpha, \alpha \to \beta \models BKI) \\
&= (\models (\models S)\,\amalg\,(\models I))\,\amalg\,(\alpha \models BKI) \\
&= (\models SI)\,\amalg\,(\alpha \models BKI)\ =\ B(SI)(BKI)
\end{aligned}$$

The overall result $B(SI)(BKI)$ is shorter than $S(S(KS)(KI))(S(KK)I)$ we obtained in Sect. 2 with the original Curry bracket abstraction – although far from being optimal. We describe the improvements in Sect. 3.1.

Figure 6, when read across, actually defines the translation process formally: each *row* of the figure gives the translation for the lambda-term of a particular

[4] The optimal solution is described in Sect. 6.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha \to \beta \vdash z : \alpha \to \beta} \quad \dfrac{\dfrac{}{\alpha \vdash z : \alpha}}{\alpha, \alpha \to \beta \vdash s\, z : \alpha}\, W}{\alpha, (\alpha \to \beta) \vdash z\ (s\, z) : \beta}}{\alpha \vdash \lambda z\ (s\, z) : (\alpha \to \beta) \to \beta}}{\vdash \lambda\, \lambda z\ (s\, z) : \alpha \to (\alpha \to \beta) \to \beta} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\alpha \to \beta \models I} \quad \dfrac{\dfrac{}{\alpha \models I}}{\alpha, \alpha \to \beta \models BKI}\, EW}{\alpha, (\alpha \to \beta) \models B(SI)(BKI)}}{\alpha \models B(SI)(BKI)}}{\models B(SI)(BKI)}$$

Fig. 7. The type and meaning derivations for the running example

form. For example, consider a term $\Gamma^+, \sigma \vdash s\, e : \tau$. (Since this section deals with the Church-style calculus, each (sub)term comes annotated with its type and typing environment.) According to the second row of Fig. 6, we have to find the translation for $\Gamma^+ \vdash e : \tau$, which is $\Gamma^+ \models d$ for some combinator expression d. Next, we apply the (EW) rule. Lambda-abstraction is the only subtlety: whether to use (Abs/EAbs) or (Abs$_0$/EAbs$_0$) depends on the type environment. This formal process is implemented in OCaml, see Sect. 4.

To show correctness, we define the reverse translation $(d)_\Lambda$ from a combinator term d to a lambda-term, by substituting S, K, I, B, S combinators with the corresponding lambda-terms and treating combinator application as lambda-application.

Theorem 1 (Translation soundness/Semantics adequacy). *Let* $\mathcal{E}[\tau_n, \ldots, \tau_1 \vdash e : \tau]$ *be* $\tau_n, \ldots, \tau_1 \models d$. *Then* $(d)_\Lambda\ s^{n-1}z\ \ldots sz\ z =_{\beta\eta} e$

This is the generalization of [10, Theorem 5.1.14] to open terms. The proof is the easy structural induction on the meaning derivation. The totality of the translation gives

Corollary 1 (Combinatorial Completeness). *Every, even open, lambda-term can be represented by a combinator term*

Or: every open term can be put in the form where all of its free variables are "on the right margin" – moreover, that form can be computed compositionally. So far, we have been dealing with the simply-typed calculus. Section 6 shows the general untyped case.

3.1 Lazy Weakening

The denotation $\tau_n, \ldots, \tau_1 \models d$ says that a combinator d should be considered in an environment, of being applied to n other terms. For example, to understand the meaning of $\mathcal{E}[\tau, \sigma \vdash s\, z : \tau]$, which is $\tau, \sigma \models K$, we have to apply K to two other terms, d_2 and d_1. This example also shows that some of these environment terms are ignored. Knowing what is ignored is useful: it leads to a better translation, as we are about to see.

To keep track of ignored context terms we mark them with the special "any type" - (analogous to the _ type placeholder in, say, OCaml). Figure 8 shows the

$$
\begin{array}{cc}
\Gamma \vdash e : \tau & \mathcal{E}\,[\Gamma \vdash e : \tau] \\
\hline
\dfrac{\Gamma^+ \vdash e : \tau}{\Gamma^+, \text{-} \vdash s\ e : \tau}\,W & \dfrac{\Gamma^+ \models d}{\Gamma^+, \text{-} \models d}\,EW \\
\dfrac{\Gamma^+, \text{-} \vdash e : \tau}{\Gamma^+ \vdash \lambda\, e : \sigma \to \tau}\,Abs_1 & \dfrac{\Gamma^+, \text{-} \models d}{\Gamma^+ \models (\models K)\amalg(\Gamma^+ \models d)}\,EAbs_1
\end{array}
$$

$$
\begin{array}{ll}
\Gamma', \text{-} \sqcup \Gamma, \text{-} \;= (\Gamma' \sqcup \Gamma), \text{-} & (\Gamma', \text{-} \models d') \amalg (\Gamma, \text{-} \models d) \;= (\Gamma' \models d') \amalg (\Gamma \models d) \\
\Gamma', \text{-} \sqcup \Gamma, \tau = (\Gamma' \sqcup \Gamma), \tau & (\Gamma', \text{-} \models d') \amalg (\Gamma, \tau \models d) = ((\models B) \amalg (\Gamma' \models d')) \amalg (\Gamma \models d) \\
\Gamma', \tau \sqcup \Gamma, \text{-} = (\Gamma' \sqcup \Gamma), \tau & (\Gamma', \tau \models d') \amalg (\Gamma, \text{-} \models d) = ((\models C) \amalg (\Gamma' \models d')) \amalg (\Gamma \models d)
\end{array}
$$

Fig. 8. The 'weak lazy leftless' type system and the denotational semantics. Shown are the differences from Fig. 6: the changed rule (W) and the new rule (Abs_1). The functions $\sqcup$ and $\amalg$ are also extended as shown. This system is presented in full in Sect. 4.

new type system and denotational semantics. The type system is clearly equivalent to that of Fig. 6. Now, the rule (EW) does nothing: ignored context terms are merely marked but ignored when computing the denotation. The real weakening is delayed until (EAbs$_1$): abstracting over an ignored variable gives the constant function. This rule corresponds to the so-called K-optimization, which ensures *full laziness* [7]. Within the bracket abstraction (Fig. 4) the optimizations is written [7, Sect. 16.2.1] as $\mathcal{A}_x\,[e] \mapsto K\,e$ iff x is not free in e. We have accomplished it *without* needing to compute and search the set of free variables. (The sequence Γ, a part of the term denotation, is all we need to know about free variables.)

$$
\dfrac{\dfrac{\dfrac{}{\alpha \to \beta \vdash z : \alpha \to \beta} \quad \dfrac{\dfrac{}{\alpha \vdash z : \alpha}}{\alpha, \text{-} \vdash s\,z : \alpha}\,W}{\alpha, (\alpha \to \beta) \vdash z\ (s\,z) : \beta}}{\vdash \lambda\lambda\, z\ (s\,z) : \alpha \to (\alpha \to \beta) \to \beta}
\qquad\qquad
\dfrac{\dfrac{\dfrac{}{\alpha \to \beta \models I} \quad \dfrac{\dfrac{}{\alpha \models I}}{\alpha, \text{-} \models I}\,EW}{\alpha, (\alpha \to \beta) \models B(CI)I}}{\models B(CI)I}
$$

Fig. 9. The type and meaning derivations for the running example, in the weak lazy leftless system (the last two (Abs) steps are compressed)

The extended $\amalg$ rules in Fig. 8 deal with applications of open terms, one of which (or both) ignore the first contextual term. For example, $(\tau_2, \tau_1 \models d') \amalg (\tau_2, \text{-} \models d)$ must be such $\bar{d}$ that $\bar{d}\ d_2\ d_1 = (d'\ d_2\ d_1)(d\ d_2)$. The latter is equal to $C(d'\ d_2)(d\ d_2)d_1$ and hence the desired $\bar{d}$ is the answer of $((\models C) \amalg (\tau_2 \models d') \amalg (\tau_2 \models d)$. It is the so-called C optimization, which the new (EW) rule forces upon us. Figure 9 shows the typing and meaning derivations for the running example. Compared to Fig. 7, the translation result is both shorter and simpler (avoiding the duplicating combinator S). Table 1 compares the two systems on more examples: tracking of ignored terms is truly a good optimization.

4 OCaml Implementation

The translation rules that previously have been written in mathematical notation are straightforward to turn into code, using the so-called tagless-final style [1,5]. This section does so, taking OCaml as the implementation language. The reason to show the code in detail is actually theoretical: to present the translation as an algebra and to argue for its correctness.

Our OCaml code embeds both the lambda and SKI simply-typed calculi as DSLs, which are specified as OCaml module signatures Lam and SKI. The abstract type (γ,α) repr represents lambda-terms; it is parameterized by the term type α and the type environment γ. It might take time to realize that the Lam signature is the precise re-statement of the left column of Figs. 6 and 8, but in the OCaml-readable notation: Lam tells the syntax (with $$ denoting an application) and the typing rules of the weak leftless system. (The operation $\Gamma' \sqcup \Gamma$ is done by unification during the type checking.)

```
module type Lam = sig
  type (γ,α) repr
  val z:    (α*γ,α) repr
  val s:    (β*γ,α) repr →
               (_*(β*γ),α) repr
  val lam:  (α*γ,β) repr → (γ,α→β) repr
  val ($$): (γ,α→β) repr →
               (γ,α) repr → (γ,β) repr
end
```

```
module type SKI = sig
  type α repr
  val kI:  (α→α) repr
  val kK:  (α→β→α) repr
  val kS:  ((α→β→δ)→(α→β)→α→δ) repr
  val kB:  ((α→β) → (δ→α) → δ→β) repr
  val kC:  ((α→β→δ) → (β→α→δ)) repr
  val ($!): (α→β) repr → α repr → β repr
end
```

In this notation, the running example is written as lam (lam (z $$ (s z))). The typed SKI calculus is likewise defined by the signature SKI (where $! is a SKI application). The Lam and SKI signatures clearly reveal the lambda and the combinator calculi to be algebras.

The accompanying code shows two implementations of the SKI signature, that is, two concrete SKI algebras. The carrier α repr of one algebra, PSKI, is set to be a string: it interprets every SKI expression as its printout. The other algebra is a SKI evaluator; its carrier is the set of OCaml values.

The lambda-to-SKI translation is also an algebra. It is an implementation of the Lam signature in terms of SKI:

```
module Conv(S:SKI) : Lam = struct
  type (γ,α) repr =   | C: α S.repr → (γ,α) repr
                      | N: (γ,α→β) repr → (α*γ,β) repr
                      | W: (γ,α) repr → (_*γ,α) repr

  let z:   (α*γ,α) repr = N (C S.kI)                      (*Var*)
  let s:   (β*γ,α) repr → (_*(β*γ),α) repr = fun e → W e  (*EW*)
```

```
    let rec ($$): type g a b. (g,a→b) repr → (g,a) repr → (g,b) repr =
     fun e1 e2 → match (e1,e2) with
      | (W e1, W e2)      → W (e1 $$ e2)   (*(se1)(se2) = s(e1 e2)*)
      | (W e, C d)        → W (e $$ C d)   (*(se)d = s(ed)*)
      | (C d, W e)        → W (C d $$ e)
      | (W e1, N e2)      → N ((C S.kB) $$ e1 $$ e2)
      | (N e1, W e2)      → N ((C S.kC) $$ e1 $$ e2)
      | (N e1, N e2)      → N ((C S.kS) $$ e1 $$ e2)
      | (C d, N e)        → N (C S.(kB $! d) $$ e)
      | (N e, C d)        → N (C S.(kC $! kC $! d) $$ e)
      | (C d1, C d2)      → C (S.(d1 $! d2))   (*closed term application*)

    let lam: (α*γ,β) repr → (γ,α→β) repr = function
      | C d → C S.(kK $! d)            (*Abs0*)
      | N e → e                        (*Abs*)
      | W e → (C S.kK) $$ e            (*Abs1*)

    let observe : (unit,α) repr → α S.repr = function (C d) → d
  end
```

This implementation re-tells the right-column of Fig. 8, the bottom-up meaning computation, spelling out II in full detail. The carrier is the GADT disjoint union of three sets: closed-term denotations (e.g., $\models I$ is written in OCaml as C S.kI), open-term denotations that ignore the context term (tagged with W) and general open-term denotations. If d of the OCaml type (γ, $\alpha \rightarrow \beta$) repr represents $\Gamma \models d$ then N d: ($\alpha * \gamma$, β) repr represents $\Gamma, \alpha \models d$. Interpreting the running example lam (lam (z $$ (s z))) in this Conv(PSKI) algebra (with the PSKI implementation of SKI) gives the string "B(CI)I".

Perhaps surprisingly, the implementation Conv makes a theoretical point. The type of observe reads as a proposition that a closed lambda-term translates to a SKI term of *the same type*. The type preservation is hence checked by the OCaml type checker. Furthermore, the OCaml compiler reports no inexhaustive pattern-match warnings: observe is hence total. The type-preservation of the translation gives, by free theorems, a certain degree of functional correctness. For example, the OCaml type checker assures that a lambda-term of the type $\alpha \rightarrow \beta \rightarrow \alpha$ is converted to a SKI term of the same type. Any term of that type (in a strongly normalizing calculus) have the same meaning. We see that not only the translation algorithm (Fig. 8) preserves the meaning but so does its implementation (the Conv functor).

4.1 The Eta-Optimization

The translation result for our running example, *B(CI)I*, shows the room for improvement. After all, B is the functional composition, of which I is the unit. Alternatively, $BdI \mapsto d$ is an η-reduction. We can implement this simplification as the second phase of the translation, in the standard tagless-final optimization framework[5]. It is more instructive to work it out into the translation itself.

[5] http://okmij.org/ftp/tagless-final/course/optimizations.html.

Just as in Sect. 3.1, procrastination is the key: delay the introduction of the I combinator. Instead of using I for the denotation of z, we add a dedicated element V to our semantic domain:

```
type (γ,α) repr = | C: α S.repr → (γ,α) repr
                  | V: (α∗γ,α) repr
                  | N: (γ,α→β) repr → (α∗γ,β) repr
                  | W: (γ,α) repr → (_∗γ,α) repr
```

with the corresponding changes to the semantic functions

```
let z:   (α∗γ,α) repr = V
let lam: type a b g. (a∗g,b) repr → (g,a→b) repr = function
  | V   → C S.kI
  ...

let rec ($$): type g a b. (g,a→b) repr → (g,a) repr → (g,b) repr =
 fun e1 e2 → match (e1,e2) with
  | (W e,V)          → N e
  | (V, W e)         → N (C (S.(kC $! kI)) $$ e)
  | (N e, V)         → N (C S.kS $$ e $$ C S.kI)
  | (V, N e)         → N (C S.(kS $! kI) $$ e)
  | (C d, V)         → N (C d)
  | (V, C d)         → N (C S.(kC $! kI $! d))
  ...
```

Just like K in Sect. 3.1, I is introduced upon abstraction and in some cases upon applications. On the other hand, when applying a closed term d to V, the free variable is already at the right margin so the denotation becomes $\tau \models d$ with no extra combinators. With this optimization, the running example translates to mere CI, which is indeed the shortest combinator for the flipped application – quite an improvement over the naive translation $S(S(KS)(KI))(S(KK)I)$ in Sect. 1. The comparison Table 1 shows the current algorithm is by far the best: e.g., it translates the K combinator lambda-term to just K and the S combinator term to S. Yet the recursion in the $$ semantic function betrays non-linear complexity. We fix the problem in Sect. 6.

5 Compiling Real Programs

The simply-typed lambda-calculus considered so far is not even Turing-complete, let alone convenient. We want numbers, booleans, convenient conditionals, and general recursion. Fortunately, all these features are easy to add, as constants (assuming a non-strict evaluation strategy) of appropriate types. Below is an example, borrowed from [7, Sect. 16.2.6]. It is the lambda-term for finding the greatest common divisor of two integers a and b (with $b \le a$) using Euclid's algorithm, written in the extended calculus embedded in OCaml:

```
let gcd = fix $$ lam (lam (lam
  (let self = s (s z) and a = s z and b = z in
   if_ $$ (eq $$ int 0 $$ b) $$ a $$ (self $$ b $$ (rem $$ a $$ b)))))
```

(The let-expression of the host language (OCaml) is the free syntax sugar that makes the term more readable.) Its translation to SKI (also extended with combinators such as Y, IF, Rem and 0) is

$$Y(B(S(BS(C(B\ IF\ (=0)))))(CC\ Rem\ (BBS)))$$

The algorithm from Sect. 4.1 was used as it is, treating fix, $\mathsf{if}\_$, etc. constants as primitive combinators. The result is compact and can be shown with no 'cheating' ([7, Sect. 16.2.6] translated only the function's body, without the recursive knot).

6 Linear algorithm

This section describes the time- and space-linear translation algorithm, in the general case of the untyped lambda-calculus (which can be backported to the typed case). We stress that for clarity the algorithm is based on the simpler Fig. 6 rather than the optimized Fig. 8 (and hence has room for improvement).

However odd, we continue to use 'type derivations', whose judgments omit types, and represent the 'type' environment Γ by its length, the natural number: e.g., $1 \vdash z$ (with the denotation $1 \models I$). The zero length is omitted. The unitype system and the corresponding denotation rules are presented in Fig. 10, which is the trivial simplification of Fig. 6.

The only interesting part is the new definition of the semantic function $\amalg$, which computes the denotation of application. Its defining requirement is

$$(n \models d')\amalg(m \models d) = \bar{d} \quad \text{iff} \quad (d'\ d_n\ \dots\ d_1)(d\ d_m\ \dots\ d_1) = \bar{d}\ d_{\max n\, m}\ \dots\ d_1$$

for some terms $d_{\max n\, m}, \dots, d_1$. Previously, in Sect. 3, such $\bar{d}$ was computed by induction on $\max n\, m$. (The two applied terms share the $\min n\, m$ suffix of their environment, which is easy to see from Fig. 6.) In contrast, Fig. 10 defines $\amalg$ in terms of so-called bulk combinators B_n, C_n and S_n, without any recursion. The bulk combinators (whose reductions and definitions in terms of S and K are collected in Fig. 11) are specifically designed to satisfy the defining requirement of $\amalg$. For example:

$$\begin{aligned}
&(d'\ d_n\ \dots\ d_1)(d\ d_n\ \dots\ d_1) = S_n\ d'\ d\ d_n\ \dots\ d_1\\
&\quad \equiv ((n \models d')\amalg(n \models d))\ d_n\ \dots\ d_1\\
&(d'\ d_{n+k}\ \dots\ d_1)(d\ d_n\ \dots\ d_1) = S_n\ (d'\ d_{n+k}\ \dots\ d_{n+1})\ d\ d_n\ \dots\ d_1\\
&\quad = (B_k\ S_n\ d'\ d_{n+k}\ \dots\ d_{n+1})\ d\ d_n\ \dots\ d_1\\
&\quad = C_k(B_k\ S_n\ d')\ d\ d_{n+k}\ \dots\ d_{n+1}\ d_n\ \dots\ d_1\\
&\quad \equiv ((n+k \models d')\amalg(n \models d))\ d_{n+k}\ \dots\ d_1 \quad k > 1
\end{aligned}$$

The bulk B_n, C_n and S_n combinators, like the ordinary B, C and S, take two terms d' and d and then $n \geq 1$ more terms and distribute the latter across the first two. The bulk combinators are thus the generalization of ordinary ones, to

$n \vdash e$	$\mathcal{E}\,[n \vdash e]$
$\dfrac{}{1 \vdash z}\,Var$	$\dfrac{}{1 \models I}\,EVar$
$\dfrac{n+1 \vdash e}{n+2 \vdash s\ e}\,W$	$\dfrac{n+1 \models d}{n+2 \models (\models K)\,\mathrm{II}\,(n+1 \models d)}\,EW$
$\dfrac{\vdash e}{\vdash \lambda e}\,Abs_0$	$\dfrac{\models d}{\models K\,d}\,EAbs_0$
$\dfrac{n+1 \vdash e}{n \vdash \lambda e}\,Abs$	$\dfrac{n+1 \models d}{n \models d}\,EAbs$
$\dfrac{n \vdash e_1 \quad m \vdash e_2}{\max n\,m \vdash e_1\ e_2}\,App$	$\dfrac{n \models d_1 \quad m \models d_2}{\max n\,m \models (n \models d_1)\,\mathrm{II}\,(m \models d_2)}\,EApp$

$$
\begin{array}{lcll}
(\models d_1)\,\mathrm{II}\,(\models d_2) & = & d_1\ d_2 & \\
(\models d_1)\,\mathrm{II}\,(n \models d_2) & = & B_n\ d_1\ d_2 & \\
(n \models d_1)\,\mathrm{II}\,(\models d_2) & = & C_n\ d_1\ d_2 & \\
(n \models d_1)\,\mathrm{II}\,(n \models d_2) & = & S_n\ d_1\ d_2 & \\
(n \models d_1)\,\mathrm{II}\,(m \models d_2) & = & B_{m-n}(S_n\ d_1)\ d_2 & \text{if } n < m \\
(n \models d_1)\,\mathrm{II}\,(m \models d_2) & = & C_{n-m}(B_{n-m}\ S_m\ d_1)\ d_2 & \text{if } n > m
\end{array}
$$

Fig. 10. The unityped system and the denotational semantics

distribute several terms 'in bulk'. (The bulk combinators are typeable and hence usable also in the typed case.)

The untyped combinator calculus permits divergent terms. The denotation of the closed $(\lambda z\, z)(\lambda z\, z)$ comes out as $(SII)(SII)$; the divergent combinator term as denotation is natural and expected. One may wish, however, denotations of open terms be convergent. To this end, one may read $n \models d$ as if it were $n \models I_n d$. The (assumed) I_n combinator ensures no reductions taking place until the full environment is supplied.

$$
\begin{array}{ll}
B'\ d\ f\ g\ x \leadsto d\ f\ (gx) & B' = BB \\
C'\ d\ f\ g\ x \leadsto d\ (fx)\ g & C' = B(BC)B \\
S'\ d\ f\ g\ x \leadsto d\ (fx)\ (gx) & S' = B(BS)B \\
 & \\
B_n\ f\ g\ x_n\ \ldots\ x_1 \leadsto f\ (g\ x_n \ldots x_1) & B_n = \mathrm{times}_{n-1}\, B'\ B \\
C_n\ f\ g\ x_n\ \ldots\ x_1 \leadsto (f\ x_n \ldots x_1)\ g & C_n = \mathrm{times}_{n-1}\, C'\ C \\
S_n\ f\ g\ x_n\ \ldots\ x_1 \leadsto (f\ x_n \ldots x_1)\ (g\ x_n \ldots x_1) & S_n = \mathrm{times}_{n-1}\, S'\ S \\
I_n\ f\ x_n\ \ldots\ x_1 \leadsto f\ x_n \ldots x_1 & I_n = B_n I \quad \text{(we set } I_0 \text{ to be } I\text{)}
\end{array}
$$

Fig. 11. Primed (director) and bulk combinators: reductions and SK definitions. The n-times application of d to d' is denoted as $\mathrm{times}_n\, d\ d'$.

In the earlier approaches, the size of the translation result has to be, in the worst case, at least quadratic in the size of the input term. This is easy to see on the worst term $\lambda x_1 \dots \lambda x_n . x_n \dots x_1$ from [7]. Its body has n variables each requiring a different component from the environment, and $n-1$ applications. Distributing one environment variable across an application requires one combinator. The bulk combinators distribute several variables in bulk, hence we expect improvement.

Let us analyze the time and space complexity of the translation. The complexity measure is the size of the input lambda term, or the number of its constructors: z, s, λ and the application. Alternatively, this is the size of the term's typing derivation, as each constructor corresponds to a rule in the derivation. If we regard B_n, C_n and S_n as pre-computed (see below) and take as the cost metric the number of combinator applications, we see from Fig. 10 that each (typing or semantic) rule contributes at most 4 combinator applications (the worst case is the application of $n \vdash e_1$ to $m \vdash e_2$ where $n > m$). We stress that II is no longer recursive. With the reasonable representation of combinator terms (e.g., as trees) an application takes fixed amount of time. Thus, the time complexity is linear. The size of the created term is also proportional to the number of applications (i.e., the internal nodes of the binary tree of the term). Therefore, the space complexity – the size of the result – is also linear in the size of the input term. The last three rows of Table 1 lists the worst-case terms of increasing size. The last column clearly shows the linear size increase for the present translation. Overall, the experience so far (including the worst-case examples of [6,7]) shows the number of combinators in the translation result stays within 1.5$\times$ the size of the input term.

The linear time- and space complexity may seem surprising. To intuitively understand it, let's apply a grossly simplified translation to $\lambda x_1 \dots \lambda x_n . e_1 \; e_2$. First we distribute the n-component environment to both e_1 and e_2 with the bulk S_n. That costs us only one combinator. If, say, e_1 is a variable x_i, it has to project one component from the environment. Realizing the projection may take $\Theta(n-i)$ combinators. However, in De Bruijn notation, x_i is encoded as $s^{n-i}z$, whose size $n-i+1$ pays for the projection combinators. If e_1 happens to be an application, we again use S_n to distribute the environment to both applicands. It costs us one combinator, which is paid by the application (which adds one to the size of the input term). The real translation uses B_n and C_n to avoid distributing unused prefixes of the environment.

We have assumed that B_n, C_n and S_n have been precomputed (as they would be in practice). If not, we have to compute them: scan the input term to determine its size n and compute and store the three sequences of bulk combinators, of n elements each. Each sequence, say, $B_i, 1 \leq i \leq n$, is built by iteration, applying the primed combinator such as B' to the previous element of the sequence. All in all, the required time and space is linear in n.

We regard arithmetic operations (such as integer comparison, subtraction, etc) and array dereference (to fetch a pre-computed bulk combinator) taking constant time – common in analyses of data structures. (All numbers in our algorithm are bound by the size of the input term. Therefore, if the size of a term does not fit within a machine register, the term does not fit into memory. Processing such terms has a very different cost model, so the traditional complexity analysis becomes pointless.)

7 Related Work

The bracket abstraction is a classical, textbook algorithm, with many descriptions and explanations and blogs[6]; we have already mentioned [7]. The algorithm and its many variations are syntactic, with typical optimization side-conditions of the form $x \notin FV(e)$. In our semantic approach, the type environment, being part of the denotation, is all we need to know about free variables. We never search through it.

The most recent work on combinator translation [9] involves so-called director strings [7, Sect. 16.3], which tell for each application node which parameters should go left or right or both ways. The bulk combinators in Fig. 11 do a similar job, but in bulk: directing whole environments rather than single variables.

Table 1. Translation of the examples using different methods and optimizations.

Lambda-term	Size	Figure 6	Figure 8	Section 4.1	Section 6
$\lambda\lambda\, z$	3	KI	KI	KI	KI
$\lambda\lambda\, sz$	4	BKI	BKI	K	BKI
$\lambda\lambda\, sz\ z$	6	CCI (BS(BKI))	CCI(BBI)	I	C(BS(BK))I
$\lambda\lambda\, z\ sz$	6	B(SI)(BKI)	B(CI)I	CI	B(SI)(BKI)
$\lambda\lambda\lambda\, z\ s^2z$	8	B(B(SI)) (B(BK)(BKI))	BK(B(CI)I)	BK(CI)	B_2(SI) (B_2K(BKI))
$\lambda\lambda\lambda(\lambda\, z)\ s^2z$	9	B(B(BI)) (B(BK)(BKI))	BK(BK(BII))	BK(BKI)[‡]	B_3I(B_2K(BKI))
$\lambda\lambda\lambda(s^2z\ z)\ (s\, z\ z)$	13	CC(CCI(BS(BKI))) (BS(B(BS)(B(CCI)(B(BS) (B(BK)(BKI))))))	CC(CCI(BBI)) (BB(BS (CCI(BBI))))	S	C(BS_2(C_2 (B_2S(B_2K (BKI)))I)) (C(BS(BKI))I)
The worst-case family for the combinator translation					
$\lambda\lambda\, z\ sz$	6	B(SI)(BKI)	B(CI)I	CI	B(SI)(BKI)
$\lambda\lambda\lambda\, z\ sz\ s^2z$	11	B(S(BS (B(SI)(BKI)))) (B(BK)(BKI))	B(C(BC (B(CI)I)))I	C(BC(CI))	B(S_2 (B(SI)(BKI))) (B_2K(BKI))
$\lambda\lambda\lambda\lambda\, z\ sz\ s^2z\ s^3z$	17	B(S(BS(B(BS) (B(S(BS(B(SI)(BKI)))) (B(BK)(BKI)))))) (B(B(BK)) (B(BK)(BKI)))	B(C(BC(B(BC) (B(C(BC (B(CI)I)))I)))))I	C(BC(B(BC) (C(BC(CI)))))	B(S_3(B(S_2 (B(SI)(BKI))) (B_2K(BKI)))) (B_3K(B_2K (BKI)))

[‡]The unoptimized (BKI) comes from a redex: the input term is not normal.

Hughes' supercombinators [3] is the alternative combinator-based implementation strategy for functional languages, tightly connected to lambda-lifting. Unlike supercombinators, which are program-specific, our bulk combinators are

[6] It is worth pointing out one, comprehensive web page: http://www.cantab.net/users/antoni.diller/brackets/intro.html.

general-purpose and can be pre-computed once and for all (or even wired into hardware). The supercombinators and the SKI translation are extensively compared in [7, Sect. 16.4], including the performance: the SKI translation time and the size of the resulting combinator term are worst-case quadratic in the size n of the (*not* De Bruijn-indexed) input lambda-term (although the typical complexity is $O(n \log n)$). In contrast, Hughes supercombinator process has the worst-case size complexity of $O(n \log n)$, but is often linear.

Our bulk combinators B_n, C_n, S_n are the same as Noshita's [6] $\bar{B_n}, \bar{C_n}, \bar{S_n}$ (but not his B_n, C_n, S_n), although introduced and used differently. (Like [6], we take each bulk combinator reference – a pointer to the pre-computed combinator sequence – to take constant space.) Noshita's combinator translation algorithm is the extension of Turner's and is syntactic. Noshita proves the $O(n \log n)$ upper-bound on the translation size; there is no claims about time complexity. Comparing ours and Noshita's approaches are difficult: not only the translation algorithms differ and give different (but equivalent) results; our calculi and input size measures also differ. Ours is lambda-calculus with De Bruijn indexes (since we also work with the typed calculus, s has to be a constructor and is counted as such). Noshita's input are binary trees with constants and named variables at the leaves – but no lambdas: Noshita only deals with supercombinators.

8 Conclusions

We have presented a semantic approach to translating lambda-terms to SKI combinators: the translation is a compositional computation of the meaning of a term. The key ideas are the choice of the semantic domain (the set of combinators) and the representation of open terms. Our presentation has stressed the parallel between type- and meaning derivations. We have demonstrated how easy, 'natural' it is to introduce various optimizations – leading all the way to the time- and space- linear translation algorithm.

The semantic approach easily extends to untyped calculus and to real programs with integers, conditionals, fixpoint, etc.

The linear translation algorithm has all the attractiveness of the supercombinator approach, but using general-purpose combinators, which can be pre-computed or even wired-in. Perhaps combinators as the compilation target of real functional languages deserve a second look.

Acknowledgments. I thank Yukiyoshi Kameyama for his challenge to write the SK conversion in the tagless-final style, and helpful discussions. I am very grateful to Doaitse Swierstra, Fritz Henglein and Noam Zeilberger for many helpful comments and discussions. Numerous suggestions by anonymous reviewers have greatly helped improve the presentation.

References

1. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009)
2. Curry, H.B., Feys, R.: Combinatory Logic. North-Holland, Amsterdam (1958)
3. Hughes, R.J.M.: Super combinators: a new implementation method for applicative languages. In: Symposium on LISP and Functional Programming, pp. 1–10. ACM, August 1982
4. Joy, M.S., Rayward-Smith, V.J., Burton, F.W.: Efficient combinator code. Comput. Lang. **10**(3/4), 211–224 (1985)
5. Kiselyov, O.: Typed Tagless Final Interpreters. In: Gibbons, J. (ed.) Generic and Indexed Programming. LNCS, vol. 7470, pp. 130–174. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_3
6. Noshita, K.: Translation of Turner combinators in O(n log n) space. Inf. Process. Lett. **20**(2), 71–74 (1985)
7. Peyton Jones, S.: The Implementation of Functional Programming Languages. Prentice Hall, Upper Saddle River, January 1987. https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/
8. Schönfinkel, M.: Über die Bausteine der mathematischen Logik. Math. Ann. **92**(3), 305–316 (1924)
9. Sinot, F.R.: Director strings revisited: a generic approach to the efficient representation of free variables in higher-order rewriting. J. Log. Comput. **15**(2), 201–218 (2005)
10. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard isomorphism. Technical report 98/14 (TOPPS note D-368), DIKU, Copenhagen (1998)
11. Stoye, W.R.: The implementation of functional languages using custom hardware. Ph.D. thesis, Computer Laboratory, University of Cambridge, December 1985. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-81.pdf
12. Turner, D.A.: Another algorithm for bracket abstraction. J. Symb. Log. **44**(2), 267–270 (1979)
13. Turner, D.A.: A new implementation technique for applicative languages. Softw.-Pract. Exp. **9**, 31–49 (1979)