Implementation of Explicit Substitutions: from $\lambda\sigma$ to the Suspension Calculus

Vincent Archambault-Bouffard
DIRO - Université de Montréal
archambv@iro.umontreal.ca

Stefan Monnier
DIRO - Université de Montréal
monnier@iro.umontreal.ca

Abstract

Explicit substitutions are an important tool for the efficient implementation of the normalization of terms in programs that manipulate data with binders, such as theorem provers or type checkers. We explore here the design space between the $\lambda\sigma$-calculus [Abadi et al. 1991] and the suspension-calculus [Nadathur and Wilson 1998] by showing how to go from one to the other, in various small steps. This gives an intuition about the performance advantages of the suspension-calculus as well as provides various alternatives.

1. Introduction

The substitution operation at the core of most formal languages, sometimes written using a notation such as $e_1[e_2/x]$, is often taken for granted but can be surprisingly tricky to formalize correctly. This motivated the development of explicit substitutions calculi such as $\lambda\sigma$ [Abadi et al. 1991] where substitution is not taken as a black box, but is instead decomposed into several primitive steps, reified explicitly as part of the syntax of terms, along with rewrite rules.

Explicit substitutions calculi are especially useful for implementing theorem provers or type checkers. In both applications, one of the driving motivations for their use is their ability to combine substitutions of various variables into a single term traversal. This makes a significant difference to the performance of operations such as term normalization and higher order unification [Liang et al. 2004].

Among the various substitutions calculi, the ones that are able to combine individual substitutions into a single term traversal fall into two large families: the variants of Abadi's $\lambda\sigma$ and the various flavors of the suspension-calculus [Nadathur and Wilson 1998]. The first is a theoretical formalism that aims to bring out the core concepts as cleanly as possible, whereas the second was developed in the context of pragmatic needs and aims to be amenable to an efficient implementation.

Our contribution is to bridge the gap between these two families, by showing how to go from one to the other, in various small steps. Each step comes with an explanation for the motivation behind it, thus creating a kind of bidirectional narrative, where one direction explains the algorithmic advantages of the suspension-calculus, while the other direction gives an intuition about how the suspensions used in the suspension-calculus can be mapped back to the core concepts in $\lambda\sigma$.

2. The $\lambda\sigma$-calculus

Our starting point is the $\lambda\sigma$ as described in [Abadi et al. 1991], although we use a slightly different notation. The calculus, like all others in this article, is using a representation of terms based on de Bruijn indices [de Bruijn 1972] so as to have a canonical representation of variables and hence avoid the cost of α-conversion. The syntax of the calculus is:

$$t ::= \#0 \mid t_1\ t_2 \mid \lambda t \mid t[\sigma]$$
$$\sigma ::= \mathsf{id} \mid t \cdot \sigma \mid \uparrow \mid \sigma_1 \circ \sigma_2$$

The syntax is split into the definition of lambda terms t and substitutions σ. In terms, $\#0$ is a de Bruijn index referencing the most recently introduced variable; $t_1\ t_2$ is a function application; λt is a lambda abstraction; and $t[\sigma]$ is a *closure* or *suspension* which stands for the substitution σ applied to the term t.

In substitutions, id is the *identity substitution* which does not substitute anything; $t \cdot \sigma$ is called a *cons* and is a substitution which replaces $\#0$ with t and applies σ to the other variables; $\uparrow$ is pronounced *shift* and is the substitution which simply increments by one the de Bruijn index of every variable reference; and $\sigma_1 \circ \sigma_2$ is the *composition* of the application of the substitution σ_1 followed by the application of the substitution σ_2. The *cons* operator $\cdot$ binds tighter than the *composition* operator $\circ$. The intuition underlying these substitution terms is well explained in the original article [Abadi et al. 1991].

Notice that this syntax represents non-zero de Bruijn indices via terms of the form $\#0[\uparrow \circ \cdots \circ \uparrow]$ with an appropriate number of $\uparrow$.

The calculus's reduction rules are shown in Fig. 1. They are split into 3 parts: the β rule which introduces *suspensions*, the *reading* rules which describe how to propagate substitutions down the terms, and the *merge* rules which describe how to reduce compositions to simpler substitution expressions. It can be shown that all the rules together correspond to the classical β-reduction rule.

In [Curien et al. 1996], the authors investigate the confluence properties of this calculus.

$$\begin{array}{lrcl}
(\beta) & (\lambda t_1)\ t_2 & \rightsquigarrow & t_1[t_2 \cdot \mathsf{id}] \\
(r_0) & \#0[t \cdot \sigma] & \rightsquigarrow & t \\
(r_\mathrm{i}) & \#0[\mathsf{id}] & \rightsquigarrow & \#0 \\
(r_\mathrm{a}) & (t_1\ t_2)[\sigma] & \rightsquigarrow & t_1[\sigma]\ t_2[\sigma] \\
(r_\mathrm{l}) & (\lambda t)[\sigma] & \rightsquigarrow & \lambda t[\#0 \cdot (\sigma \circ \uparrow)] \\
(r_\mathrm{s}) & t[\sigma_1][\sigma_2] & \rightsquigarrow & t[\sigma_1 \circ \sigma_2] \\
(m_\mathrm{idl}) & \mathsf{id} \circ \sigma & \rightsquigarrow & \sigma \\
(m_\mathrm{idr}) & \uparrow \circ \mathsf{id} & \rightsquigarrow & \uparrow \\
(m_\mathrm{shift}) & \uparrow \circ (t \cdot \sigma) & \rightsquigarrow & \sigma \\
(m_\mathrm{ass}) & (\sigma_1 \circ \sigma_2) \circ \sigma_3 & \rightsquigarrow & \sigma_1 \circ (\sigma_2 \circ \sigma_3) \\
(m_\mathrm{map}) & (t \cdot \sigma_1) \circ \sigma_2 & \rightsquigarrow & t[\sigma_2] \cdot (\sigma_1 \circ \sigma_2)
\end{array}$$

Figure 1. Reduction rules of the original $\lambda\sigma$-calculus

3. Implementing Explicit Substitutions

That formalization of $\lambda\sigma$-calculus does not lend itself immediately to an implementation, for the following reasons:

$$
\begin{array}{lrcl}
 & t ::= & c \mid v \mid \#i \mid t_1\ t_2 \mid \lambda t \mid t[\sigma] & \\
(\beta) & (\lambda t_1)\ t_2 & \rightsquigarrow & t_1[\mathit{cons}\ t_2\ \mathit{id}] \\
(l) & \#i[\sigma] & \rightsquigarrow & \mathit{lookup}\ \#i\ \sigma \\
(r_{\mathrm{c}}) & c[\sigma] & \rightsquigarrow & c \\
(r_{\mathrm{i}}) & t[\mathit{id}] & \rightsquigarrow & t \\
(r_{\mathrm{a}}) & (t_1\ t_2)[\sigma] & \rightsquigarrow & t_1[\sigma]\ t_2[\sigma] \\
(r_{\mathrm{l}}) & (\lambda t)[\sigma] & \rightsquigarrow & \lambda t[\mathit{lift}\ \sigma] \\
(r_{\mathrm{s}}) & t[\sigma_1][\sigma_2] & \rightsquigarrow & t[\mathit{compose}\ \sigma_1\ \sigma_2]
\end{array}
$$

Figure 2. Term syntax and generic rewrite rules

- The use of $\#0[\uparrow \circ \cdots \circ \uparrow]$ to access any other variable than $\#0$ is an inconvenient and inefficient representation. It also implies that you cannot hope to eliminate all $t[\sigma]$ from the code. So in practice, we will extend the syntax of variable references to $\#i$.
- The calculus lacks some elements present in many real languages, such as constants and meta-variables. To account for those needs we add c and v to our lambda terms to represent respectively constants and meta-variables.
- The calculus is defined as a set of rewrite rules without specifying any order of application of those rules.

To address those issues and stay as general as possible, Fig. 2 shows the syntax of terms and the generic rewrite rules we will use for the rest of this article. The representation of the substitutions is kept abstract in the syntax and the generic rewrite rules are correspondingly parameterized over 5 functions defined below. This will hence let us reuse the same term syntax and rewrite rules with the various explicit substitutions we present later on.

These generic definitions basically corresponds to the terms and *reading* rules of $\lambda\sigma$-calculus adapted for variables other than $\#0$, constant and meta-variables. There is no new generic *reading* rule for v terms because reduction simply cannot proceed until those variables are instantiated.

As said above, the rules are parameterized over 5 functions:

- *id*: The representation of the identity substitution.
- *cons*: Extend a substitution such that it additionally replaces $\#0$ with a given term.
- *lookup*: Apply a substitution to a variable reference. Substitutions are normalization's moral equivalent of the environments used in interpreters, which is why we like to think of this operation as looking up a variable in an environment.
- *lift*: Adjust a substitution for use inside a new binding.
- *compose*: Take two substitutions and build a new one which applies them in sequence.

The generic rewrite rules solve the first two issues described at the beginning of this section. As for the third one, while the *reading* rules are still non-deterministic, the *merge* rules are contained within the 5 functions and therefore have become deterministic.

Notice that many of the rules do not pay much attention to the substitution that is manipulated. The goal for the rest of the paper is to show how to implement the *lookup*, *lift* and *compose* functions efficiently to allow multiple substitutions to be performed in a single term traversal. This will allow us to go from the $\lambda\sigma$ to the suspension calculus.

3.1 Avoiding Composition

Before we start to optimize the $\lambda\sigma$, we would like to make another remark concerning the generic rewrite rules. If we look at the rules in Fig. 2, we see that *compose* is only explicitly needed to handle

$$
\begin{array}{lcl}
\mathit{id} & = & \mathsf{id} \\
\mathit{cons}\ t\ \sigma & = & t \cdot \sigma \\
\mathit{lift}\ \sigma & = & \#0 \cdot (\sigma \circ \uparrow) \\
\mathit{compose}\ \sigma_1\ \sigma_2 & = & \sigma_1 \circ \sigma_2
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{lookup}\ \#i\ \mathsf{id} & = & \#i \\
\mathit{lookup}\ \#0\ (t \cdot \sigma) & = & t \\
\mathit{lookup}\ \#i\ (t \cdot \sigma) & = & \mathit{lookup}\ \#(i-1)\ \sigma \\
\mathit{lookup}\ \#i \uparrow & = & \#(i+1) \\
\mathit{lookup}\ \#i\ (\mathsf{id} \circ \sigma) & = & \mathit{lookup}\ \#i\ \sigma \\
\mathit{lookup}\ \#i\ (\uparrow \circ \sigma) & = & \mathit{lookup}\ \#(i+1)\ \sigma \\
\mathit{lookup}\ \#0\ ((t \cdot \sigma_1) \circ \sigma_2) & = & t[\sigma_2] \\
\mathit{lookup}\ \#i\ ((t \cdot \sigma_1) \circ \sigma_2) & = & \mathit{lookup}\ \#(i-1)\ (\sigma_1 \circ \sigma_2) \\
\mathit{lookup}\ \#i\ ((\sigma_1 \circ \sigma_2) \circ \sigma_3) & = & \mathit{lookup}\ \#i\ (\sigma_1 \circ (\sigma_2 \circ \sigma_3))
\end{array}
$$

Figure 3. Implementation of $\lambda\sigma$

terms of the form $t[\sigma_1][\sigma_2]$. Looking further at the rules, we can see that such terms can be introduced only in the following cases:

1. One of the *reading* rules when a subterm is already itself a suspension.
2. The β rule when t_1 is itself a suspension.

When performing the typical normalization by reduction of outermost redexes, the *reading* rules should only be applied to terms that haven't been visited yet, so we have the property that there should never be a suspension $t'[\sigma']$ inside the t of another suspension $t[\sigma]$. In other words, the first case should never happen. Similarly, in the normalization case, the second case should only occur if the suspension t_1 was pushed earlier from outside the λ by the r_{lam} rule.

As it turns out, we can add a special β' rule to handle the second case without resorting to *compose*:

$$(\beta') \quad (\lambda t_1)[\sigma]\ t_2 \ \rightsquigarrow\ t_1[\mathit{cons}\ t_2\ \sigma]$$

This rule was already suggested in [Abadi et al. 1991] and a similar rule can be found in the λj calculus [Accattoli and Kesner 2010]. Note that the β' rule does not rule out the need of a compose function if one of the other 4 functions makes use of it internally.

3.2 Implementation Suggested for $\lambda\sigma$-calculus

In their paper, [Abadi et al. 1991] propose an implementation strategy for weak head normalization. We can define our 5 functions to correspond to their implementation strategy as shown in Fig. 3.

4. $\lambda\sigma_0$: Restricted $\lambda\sigma$-calculus

The $\lambda\sigma$-calculus presented so far has a very efficient implementation of *compose* but at the cost of a rather slow and complex implementation of *lookup*. Furthermore a large part of the computation performed in *lookup* risks being executed repeatedly for the same substitution applied to several variable occurrences.

Another problem is that sequences of $\uparrow$ tend to arise in many scenarios. For example, it can be shown [Abadi et al. 1991] that the normal form of a substitution in the $\lambda\sigma$-calculus is of the form $t_1 \cdot \ldots \cdot t_n \cdot (\uparrow \circ \cdots \circ \uparrow)$. Thus most substitutions will contain a sequence of $\uparrow$, and we would like to represent them more efficiently.

For these reasons, in Twelf [Pfenning and Schürmann 1999], the implementors decided to use a restricted form of the $\lambda\sigma$-calculus which we call $\lambda\sigma_0$ where the composition is not part of the syntax of substitutions but is implemented as a function instead and where $\uparrow$ is extended to $\uparrow_n$ so multiple shifts can be combined into a single one.

$$\sigma \ ::= \ t \cdot \sigma \mid \uparrow_n$$

$$\begin{array}{lcl}
\mathit{id} & = & \uparrow_0 \\
\mathit{cons}\ t\ \sigma & = & t \cdot \sigma \\
\mathit{lift}\ \sigma & = & \#0 \cdot (\mathit{compose}\ \sigma \uparrow_1) \\
\\
\mathit{lookup}\ \#0\ (t \cdot \sigma) & = & t \\
\mathit{lookup}\ \#i\ (t \cdot \sigma) & = & \mathit{lookup}\ \#(i-1)\ \sigma \\
\mathit{lookup}\ \#i\ \uparrow_n & = & \#(i+n) \\
\\
\mathit{compose}\ (\uparrow_0)\ \sigma & = & \sigma \\
\mathit{compose}\ (\uparrow_{n_1})\ (\uparrow_{n_2}) & = & \uparrow_{n_1+n_2} \\
\mathit{compose}\ (\uparrow_n)\ (t \cdot \sigma) & = & \mathit{compose}\ (\uparrow_{n-1})\ \sigma \\
\mathit{compose}\ (t \cdot \sigma_1)\ \sigma_2 & = & t[\sigma_2] \cdot (\mathit{compose}\ \sigma_1\ \sigma_2)
\end{array}$$

Figure 4. Syntax and functions of the $\lambda\sigma_0$-calculus

The corresponding syntax and instantiation of the parameters of the reading rules are shown in Fig. 4. As can be seen, the resulting syntax is pleasantly simplified and the implementation is also streamlined. Substitutions are now represented as singly-linked lists where the terminating "nil" element carries the number of shifts to apply to the other variables.

lookup now is efficient, but at the cost of more expensive *compose*. Furthermore, *compose* is now used internally for *lift*, so the rule β' is not sufficient to make the performance of *compose* irrelevant.

5. $\lambda\sigma_1$: Shifting Lazily

The performance of the $\lambda\sigma_0$-calculus tends to suffer from the following two issues:

- The call to *compose* in *lift* will always fall through to the last case of the *compose* function, which recurses until the end of the substitution. So it takes time proportional to the length of the substitution, and since *lift* increases the length of the substitution by 1, a sequence of N applications of *lift* has an $O(N^2)$ complexity. It would be preferable to try and push those $\uparrow$ to the leaves more lazily, so we can combine them along the way.
- Another related problem comes when we need to compute something of the form $(t_1 \cdot \sigma_1 \circ \uparrow_n) \circ t_2 \cdot \sigma_2$. Twelf will reduce it as follows:

$$\begin{array}{ll}
& (t_1 \cdot \sigma_1 \circ \uparrow_n) \circ t_2 \cdot \sigma_2 \\
\rightsquigarrow & (t_1[\uparrow_n] \cdot (\sigma_1 \circ \uparrow_n)) \circ t_2 \cdot \sigma_2 \\
\rightsquigarrow & (t_1[\uparrow_n] \cdot ...) \circ t_2 \cdot \sigma_2 \\
\rightsquigarrow & t_1[\uparrow_n][t_2 \cdot \sigma_2] \cdot (... \circ t_2 \cdot \sigma_2) \\
\rightsquigarrow & t_1[\uparrow_n \circ t_2 \cdot \sigma_2] \cdot (... \circ t_2 \cdot \sigma_2) \\
\rightsquigarrow & t_1[\uparrow_{n-1} \circ \sigma_2] \cdot (... \circ t_2 \cdot \sigma_2)
\end{array}$$

  whereas we would want to get rid of t_2 right from the start by reducing in a different order, such as:

$$\begin{array}{ll}
& (t_1 \cdot \sigma_1 \circ \uparrow_n) \circ t_2 \cdot \sigma_2 \\
\rightsquigarrow & (t_1 \cdot \sigma_1) \circ (\uparrow_n \circ t_2 \cdot \sigma_2) \\
\rightsquigarrow & (t_1 \cdot \sigma_1) \circ (\uparrow_{n-1} \circ \sigma_2)
\end{array}$$

We can solve those two problems by using the following tweak to Twelf's approach: replace the $\uparrow$ which just does a shift, with $\uparrow\sigma$ which is a new syntax whose meaning is equivalent to $\sigma \circ \uparrow$ in the $\lambda\sigma$-calculus. Fig. 5 shows the resulting syntax and implementation.

Notice that we had to re-introduce an id substitution. More importantly, notice that the rule for *compose* σ_1 $(\uparrow_n \sigma_2)$ and the rule for *compose* $(t \cdot \sigma_1)$ σ_2 overlap, and we use the definition's ordering to make sure that we always use the first, which hoists the *shifts* outward, in preference to the second, which pushes the *shifts* deeper and duplicates them.

$$\sigma \ ::= \ \mathsf{id} \mid t \cdot \sigma \mid \uparrow_n \sigma$$

$$\begin{array}{lcl}
\mathit{id} & = & \mathsf{id} \\
\mathit{cons}\ t\ \sigma & = & t \cdot \sigma \\
\mathit{lift}\ \sigma & = & \#0 \cdot \uparrow_1 \sigma \\
\\
\mathit{lookup}\ \#i\ \sigma & = & \mathit{lookup'}\ 0\ \#i\ \sigma \\
\\
\mathit{lookup'}\ o\ \#i\ \mathsf{id} & = & \#(i+o) \\
\mathit{lookup'}\ o\ \#i\ (\uparrow_n \sigma) & = & \mathit{lookup'}\ (o+n)\ \#i\ \sigma \\
\mathit{lookup'}\ o\ \#0\ (t \cdot \sigma) & = & \mathsf{if}\ o = 0\ \mathsf{then}\ t\ \mathsf{else}\ t[\uparrow_o \mathsf{id}] \\
\mathit{lookup'}\ o\ \#i\ (t \cdot \sigma) & = & \mathit{lookup'}\ o\ \#(i-1)\ \sigma \\
\\
\mathit{compose}\ \sigma\ \mathsf{id} & = & \sigma \\
\mathit{compose}\ \sigma_1\ (\uparrow_n \sigma_2) & = & \uparrow_n (\mathit{compose}\ \sigma_1\ \sigma_2) \\
\mathit{compose}\ (\uparrow_1 \sigma_1)\ (t \cdot \sigma_2) & = & \mathit{compose}\ \sigma_1\ \sigma_2 \\
\mathit{compose}\ (\uparrow_n \sigma_1)\ (t \cdot \sigma_2) & = & \mathit{compose}\ (\uparrow_{n-1} \sigma_1)\ \sigma_2 \\
\mathit{compose}\ (t \cdot \sigma_1)\ \sigma_2 & = & t[\sigma_2] \cdot (\mathit{compose}\ \sigma_1\ \sigma_2)
\end{array}$$

Figure 5. Syntax and functions of the $\lambda\sigma_1$-calculus

One way to think about it is that the $\lambda\sigma_1$-calculus tries to keep the shifts close to the top of the substitution, so they can quickly cancel out a *cons*. This is done in the second case of the compose function. In contrast Twelf's approach always pushes the shifts all the way to the bottom of the substitutions, so we have to do more work before we can cancel a *cons* with a *shift*. Of course, there is no free lunch: we still have to propagate the *shifts* to the very bottom sooner or later, but we do it in the *lookup* rules instead, where the parameter o keeps track of how many *shifts* we have encountered along the way.

6. $\lambda\sigma_2$: Fast Lookup of Free Variables

One inefficiency in $\lambda\sigma_1$ is the treatment of *lookup* for free variables that are outside the scope of the substitution. More specifically, every substitution has a *length*, which is the number of *cons* elements. Looking at the *lookup'* function in Fig. 5, one can see that any variable reference $\#i$ whose i is larger than this length will end up in the id case and be turned into $\#(i-l+o)$ where l is the length of the environment and o is the number of *shifts* in the substitution. In other words, for any $\#i$ where i is bigger than the length of σ, *lookup'* will traverse the complete σ, only to collect the number of *cons* and the number of *shifts*.

For example, the term $(\lambda\#3)\ t_1$ will be reduced to $\#3[t_1 \cdot \mathsf{id}]$. The result is $\#2$ which corresponds to collecting 1 *cons* and 0 *shift*. As another example the term $(\lambda\lambda\#3)\ t_1$ will be reduced to $\lambda\#3[\#0 \cdot \uparrow_1 (t_1 \cdot \mathsf{id})]$. The result is $\lambda\#2$ which corresponds to collecting 2 *cons* and 1 *shift*.

The $\lambda\sigma_2$-calculus addresses this inefficiency by keeping track of the length and the total number of shifts in substitutions. More specifically, substitutions are now defined as triplets (ol, nl, e) where ol is the length of e, nl is the number of shifts in it, and e is the "raw" substitution, which has the same shape as the substitutions of $\lambda\sigma_1$. The syntax and implementation of the $\lambda\sigma_2$-calculus can be found in Fig. 6. The *lookup'* function is identical to the one for $\lambda\sigma_1$, except that $\uparrow_o \mathsf{id}$ is replace by $(0, o, \uparrow_o \mathsf{id})$.

The way $\lambda\sigma_2$-calculus keeps track of the length of the substitution and the number of shifts might not be obvious to readers unfamiliar with the suspension calculus. The *id* substitution obviously has 0 *cons* and 0 *shifts*. The *cons* function increases by one the variable ol because one more term is added to the substitution. Recall that this rule is triggered by the β-rule. The function *lift*, in-

$$\begin{array}{rcl} \sigma & ::= & (ol, nl, e) \\ e & ::= & \mathsf{id} \mid t \cdot e \mid \uparrow_n e \end{array}$$

$$\begin{array}{lcl} \mathit{id} & = & (0, 0, \mathsf{id}) \\ \mathit{cons}\ t\ (ol, nl, e) & = & (ol+1, nl, t \cdot e) \\ \mathit{lift}\ (ol, nl, e) & = & (ol+1, nl+1, \#0 \cdot \uparrow_1 e) \end{array}$$

$$\begin{array}{l} \mathit{lookup}\ \#i\ (ol, nl, e) = \\ \quad \mathsf{if}\ (i \geq ol)\ \mathsf{then} \\ \quad\quad \#(i - ol + nl) \\ \quad \mathsf{else} \\ \quad\quad \mathit{lookup'}\ 0\ \#i\ e \end{array}$$

$$\begin{array}{lcl} \mathit{compose}\ \sigma\ \mathit{id} & = & \sigma \end{array}$$

$$\begin{array}{l} \mathit{compose}\ (ol_1, nl_1, e_1)\ (ol_2, nl_2, \uparrow_n e_2) = \\ \quad (ol', nl', \uparrow_n (\mathit{compose}\ e_1\ e_2)) \\ \quad \mathsf{where}\ \ ol' = ol_1 + max(0, (ol_2 - nl_1)) \\ \qquad\qquad\ nl' = nl_2 + max(0, (nl_1 - ol_2)) \end{array}$$

$$\begin{array}{l} \mathit{compose}\ (ol_1, nl_1, \uparrow_1 e_1)\ (ol_2, nl_2, t \cdot e_2) = \\ \quad \mathit{compose}\ (ol_1, nl_1 - 1, e_1)\ (ol_2 - 1, nl_2, e_2) \end{array}$$

$$\begin{array}{l} \mathit{compose}\ (ol_1, nl_1, \uparrow_n e_1)\ (ol_2, nl_2, t \cdot e_2) = \\ \quad \mathit{compose}\ (ol_1, nl_1 - 1, \uparrow_{n-1} e_1)\ (ol_2 - 1, nl_2, e_2) \end{array}$$

$$\begin{array}{l} \mathit{compose}\ (ol_1, nl_1, t \cdot \sigma_1)\ (ol_2, nl_2, e_2) = \\ \quad (ol', nl', t[\sigma_2] \cdot (\mathit{compose}\ (\ ol_1 - 1, nl_1, \sigma_1)(ol_2, nl_2, e_2))) \\ \quad \mathsf{where}\ \ ol' = ol_1 + max(0, (ol_2 - nl_1)) \\ \qquad\qquad\ nl' = nl_2 + max(0, (nl_1 - ol_2)) \end{array}$$

Figure 6. Syntax and functions of the $\lambda\sigma_2$-calculus

creases both ol and nl by one because it adds the term $\#0$ at the front of the substitution and lifts the remaining part of the substitution. This rule is triggered by the r_{i}-rule when substitution go under a lambda.

The tricky part is how compose computes the new ol and nl for the substitution it generates. When a *shift* cancels a *cons*, compose calls itself recursively with substitutions that contains one fewer *shift* and one fewer *cons*. But when compose creates a new substitution, it computes its new length to $ol' = ol_1 + max(0, (ol_2 - nl_1))$ and its new number of *shifts* to $nl' = nl_2 + max(0, (nl_1 - ol_2))$. To see that this is correct, the reader should recall that ol and nl are used to adjust the indices of the free variables. Also, compose is used for terms like $t[\sigma_1][\sigma_2]$. When is a variable free when t is the target of two substitution σ_1 and σ_2? First it must be free for the first substitution σ_1. So its index must be greater than the length of σ_1.

$$i > ol_1 \tag{1}$$

After the first substitution is applied, the variable now has the index:

$$i' = i - ol_1 + nl_1 \tag{2}$$

Now when the second substitution is applied, in order to be free the index i' must again be greater than the length of σ_2. So we have

$$i' = i - ol_1 + nl_1 > ol_2 \tag{3}$$

Because the composition of σ_1 and σ_2 must give the same results as applying both substitution in order, its ol' is $max(ol_1, ol_1 + ol_2 - nl_1)$ which can be written $ol_1 + max(0, ol_2 - nl_1)$. The max function is used because both equations 1 and 3 must be satisfied.

For the parameter nl', we must have that the new variable index i'' after the second substitution σ_2 satisfies

$$i'' = i' - ol_2 + nl_2 = i - ol_1 + nl_1 - ol_2 + nl_2 = i - ol' + nl' \tag{4}$$

Again, the equation is justified because the composition of σ_1 and σ_2 must give the same results as applying both substitution in order.

By having $ol' = ol_1 + max(0, ol_2 - nl_1)$, it is easy to check that $nl' = nl_2 + max(0, nl_1 - ol_2)$.

The strength of $\lambda\sigma_2$-calculus comes from the fact that we can compute nl and ol when traversing a term even in the presence of substitutions composition and therefore always lookup the free variables of a term in constant time. We now turn to the suspension-calculus to improve the lookup of bound variables.

7. The Suspension Calculus

The reader familiar with the suspension calculus will notice a strong resemblance to the $\lambda\sigma_2$-calculus. To get to the suspension calculus we need to perform one last optimization. We would like to be able to lookup bound variable efficiently, without having to traverse the substitution. In $\lambda\sigma_2$-calculus we need to do so for two reasons. First, for a bound variable $\#i$, it is impossible to locate directly the position of the i^{th} *cons* element in the data structure of σ. Indeed, $t \cdot \sigma$ terms are mixed with $\uparrow_n \sigma$. Second, even if we had a way to directly access the i^{th} *cons* element, we need to collect the number of *shifts*. The only way to do so is by traversing the list.

To lift the first restriction, the terms e can be made more compact by having each step be a combination of a *cons* with some number of *shifts*. More specifically, let's consider the typical shape of an environment e in the $\lambda\sigma_2$-calculus:

$$\uparrow_{n_1}(t_1 \cdot \uparrow_{n_2}(t_2 \cdot \ldots \cdot \uparrow_{n_m} \mathsf{id}))$$

We can replace every $\uparrow_n (t \cdot \sigma)$ with a new syntax $(t, n) :: \sigma$. This syntax merges $t \cdot \sigma$ and $\uparrow_n \sigma$ together. Also, considering the way *lookup* is implemented, we can see that the final $\uparrow_{n_m} \mathsf{id}$ is not really needed, since either *lookup* only uses ol and nl or $\mathit{lookup'}_{\sigma_1}$ necessarily stops before reaching the end. We can replace $\uparrow_{n_m} \mathsf{id}$ by a constant nil.

$$(t_1, n_1) :: (t_2, n_2) :: \cdots :: \mathsf{nil}$$

We now have a substitution represented as a list. Now *lookup* can be implemented efficiently. Not only does it skip the $O(N)$ traversal for free-variables, but the traversal can use a simple indexation operation to directly fetch the n^{th} element. So e could be implemented with any standard data structure such as an array or a balanced tree.

To lift the second restriction, the suspension calculus makes one more tweak to that representation: in order to avoid having to count all the *shifts* by traversing e, the $(t, n) :: \sigma$ representation stores in n the difference between the number of shifts and the variable nl of the substitution. In other words, we have the following equivalence:

$$\begin{array}{c} (t_1, n_1) :: (t_2, n_2) :: \cdots :: (t_m, n_m) :: \mathsf{nil} \\ \simeq \\ \uparrow_{(nl - n_1)}(t_1 \cdot \uparrow_{(n_1 - n_2)}(t_2 \cdot \ldots \cdot \uparrow_{(n_{m-1} - n_m)} t_m \cdot \uparrow_{n_m} \mathsf{id})) \end{array}$$

So to get the total number of *shifts* for the variable $\#i$, we must compute $nl - n_i$. It is important to understand that the difference n_i is computed when the element t_i is added to the substitution. If the counter nl continues to grow after that, the relation is still valid because it means that new *shifts* were added to the front of the substitution and thus need to be added to the total of *shifts* collected before we reach t_i if we make a traversal like in $\lambda\sigma_2$.

7.1 Restricted Suspension Calculus

Figure 7 shows the syntax and implementation of the calculus with the new way to represent the substitution.

This version of the suspension calculus is more rudimentary than any of the published ones. But it corresponds fairly closely to the restricted suspension calculus presented in [Nadathur 1999] and used in [Shao et al. 1998].

The main difference is the absence of the $@n$ form. This form is used in the suspension-calculus as a shorthand for $(\#0, n +$

$$\begin{array}{rcl} \sigma & ::= & (ol, nl, e) \\ e & ::= & \mathsf{nil} \mid (t, n) :: e \end{array}$$

$$\begin{array}{lcl} \mathit{id} & = & (0, 0, \mathsf{nil}) \\ \mathit{cons}\ t\ (ol, nl, e) & \rightarrow & (ol+1, nl, (t, nl) :: e) \\ \mathit{lift}\ (ol, nl, e) & \rightarrow & (ol+1, nl+1, (\#0, nl+1) :: e) \end{array}$$

$$\begin{array}{l} \mathit{lookup}\ \#i\ (ol, nl, e) = \\ \quad \mathsf{if}\ (i \geq ol)\ \mathsf{then} \\ \quad\quad \#(i - ol + nl) \\ \quad \mathsf{else} \\ \quad\quad \mathsf{let}\ (t, n) = \mathit{nth}\ i\ e \\ \quad\quad \mathsf{in\ case}\ t\ \mathsf{in} \\ \quad\quad\quad t'[(ol', nl', e')] \Rightarrow t'[(ol', nl' + nl - n, e')] \\ \quad\quad\quad \_ \Rightarrow t[(0, nl - n, \mathsf{nil})] \end{array}$$

Figure 7. Syntax and functions of λ_{susp}

$$\begin{array}{l} \mathit{compose}\ (ol_1, nl_1, e_1)\ (ol_2, nl_2, e_2) = \\ \quad (ol', nl', \mathit{comp}_e\ e_1\ nl_1\ ol_2\ e_2) \\ \quad \mathsf{where}\ ol' = ol_1 + max(0, (ol_2 - nl_1)) \\ \qquad\quad\ nl' = nl_2 + max(0, (nl_1 - ol_2)) \end{array}$$

$$\mathit{comp}_e\ e_1\ nl_1\ 0\ \mathsf{nil}\ =\ e_1$$

$$\mathit{comp}_e\ \mathsf{nil}\ 0\ ol_2\ e_2\ =\ e_2$$

$$\begin{array}{l} \mathit{comp}_e\ \mathsf{nil}\ nl_1\ ol_2\ ((t, n) :: e_2) = \\ \quad \mathit{comp}_e\ \mathsf{nil}\ (nl_1 - 1)\ (ol_2 - 1)\ e_2, \\ \quad \mathsf{if}\ nl_1 \geq 1 \end{array}$$

$$\begin{array}{l} \mathit{comp}_e\ ((t_1, n_1) :: e_1)\ nl_1\ ol_2\ ((t_2, n_2) :: e_2) = \\ \quad \mathit{comp}_e\ ((t_1, n_1) :: e_1)\ (nl_1 - 1)\ (ol_2 - 1)\ e_2 \\ \quad \mathsf{if}\ nl_1 > n_1 \end{array}$$

$$\begin{array}{l} \mathit{comp}_e\ ((t_1, n_1) :: e_1)\ nl_1\ ol_2\ ((t_2, n_2) :: e_2) = \\ \quad \mathsf{let}\ e' = \mathit{comp}_e\ e_1\ n_1\ ol_2\ ((t_2, n_2) :: e_2) \\ \quad \mathsf{in}\ (t_1[(ol_2, n_2, (t_2, n_2) :: e_2)], n_2 + max(0, (n_1 - ol_2))) :: e' \end{array}$$

Figure 8. Compose function for the simplified λ_{susp}

1), which is what we use here in the *lift* function instead. The advantage of using a special form like $@n$ is to distinguish the case where a variable is replaced with another variable from the case where a variable's index is simply renumbered. This difference is significant when we care to preserve information such as the location of the variable in the source code.

The reader will probably have noticed the absence of a *compose* function, which simply reflects the fact that this calculus relies on the β' rule to avoid most uses of *compose*. And it comes with its own additional ad-hoc rule, hidden within the *lookup* function: before returning a new suspension shifted by $nl - n$, the function checks to see if the base form t is itself a suspension, in which case it performs the composition of the base substitution with the shift by $nl - n$, to avoid the need for a complete implementation of *compose*.

7.2 Simplified Suspension Calculus

Of course, if needed, we can define *compose*, for example by taking the rewrite rules presented in [Gacek and Nadathur 2007] and implementing them as a function as shown in Fig.8.

We can see in Fig. 8 that compose is similar to the one defined for $\lambda\sigma_2$. The resulting composition has the same ol' and nl'. The first two cases of comp_e correspond to the *id* case. The two other cases when a *shift* cancels a *cons*. Remember that nil replaces $\uparrow_{n_m}$id so the substitution can still represent *shifts* in the presence of nil. The last one corresponds to the last case of compose in $\lambda\sigma_2$ where the second substitution is distributed over the cons of the first.

8. Related Work

This is not the first time explicit substitutions calculi are compared. For example, Gacek and Nadathur [Gacek and Nadathur 2007] provide a formal mapping between the simplified suspension calculus and $\lambda\sigma$ as well as many other calculi. They also provide a full treatment of the properties of the simplified suspension calculus and their mathematical proofs. Our intention here is to try and give a more intuitive and pragmatically motivated mapping, going into several motivated steps instead of giving a formal mapping.

9. Conclusion

In this paper we showed how the suspension calculus can be viewed as an optimized implementation of the original $\lambda\sigma$. We hope this can motivate the use of the suspension calculus as an implementation tool even though it is not easy to figure out how it works based on its own original rules. Indeed, like most optimized algorithms, trying to understand the inner working of the suspension calculus directly can be hard. It is simpler to understand $\lambda\sigma$ and how to proceed to optimize it.

10. Acknowledgment

We would like to thank the reviewers for giving us various pointers to similar work in the litterature.

References

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

B. Accattoli and D. Kesner. The structural λ-calculus. In *Computer Science Logic*, pages 381–395. Springer, 2010.

P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2):362–397, 1996.

N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Math.*, 34(5):381–392, 1972.

A. Gacek and G. Nadathur. A simplified suspension calculus and its relationship to other explicit substitution calculi. Technical Report 2007/39, Digital Technology Center, University of Minnesota, 2007.

C. Liang, G. Nadathur, and X. Qi. Choices in representation and reduction strategies for lambda terms in intensional contexts. *Journal of Automated Reasoning*, 33:89–132, 2004.

G. Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), 1999.

G. Nadathur and D. S. Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, May 1998.

F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, July 1999.

Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *International Conference on Functional Programming*, pages 313–323. ACM Press, Sept. 1998.