Improvements for Free

Daniel Seidel[*] Janis Voigtländer
Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik
Römerstraße 164, 53117 Bonn, Germany
`{ds,jv}@informatik.uni-bonn.de`

"Theorems for Free!" (Wadler 1989) is a slogan for a technique that allows to derive statements about functions just from their types. So far, the statements considered have always had a purely extensional flavor: statements relating the value semantics of program expressions, but not statements relating their runtime (or other) cost. Here we study an extension of the technique that allows precisely statements of the latter flavor, by deriving quantitative theorems for free. After developing the theory, we walk through a number of example derivations. Probably none of the statements derived in those simple examples will be particularly surprising to most readers, but what *is* maybe surprising, and at the very least novel, is that there is a general technique for obtaining such results on a quantitative level in a principled way. Moreover, there is good potential to bring that technique to bear on more complex examples as well. We turn our attention to short-cut fusion (Gill et al. 1993) in particular.

1 Introduction

Based on the concept of relational parametricity (Reynolds 1983), Wadler (1989) established so-called "free theorems", a method for obtaining proofs of program properties from parametrically polymorphic types in purely functional languages. For example, it can thus be shown that every function $f :: [\alpha] \to [\alpha]$, with α a type variable, satisfies

$$f\ (mapList\ g\ xs) = mapList\ g\ (f\ xs) \tag{1}$$

for every choice of $g :: \tau_1 \to \tau_2$ and $xs :: [\tau_1]$, with τ_1 and τ_2 concrete types, where:

$$\begin{aligned}
&mapList :: (\alpha \to \beta) \to [\alpha] \to [\beta]\\
&mapList\ g\ [\,] \qquad\ \ = [\,]\\
&mapList\ g\ (x : xs) = (g\ x) : (mapList\ g\ xs)
\end{aligned}$$

Statements of that flavor have been used for program transformation (Gill et al. 1993; Svenningsson 2002; Voigtländer 2009a), but also for other interesting results (Voigtländer 2008; Bernardy et al. 2010a).

So far, free theorems have been considered a qualitative tool only. That is, statements like (1) have been established as extensional equivalences or semantic approximations in a definedness order, and in fact a lot of research has gone into what definedness and/or strictness conditions are needed on the involved functions in various language settings and into extending the approach to richer type systems (Launchbury and Paterson 1996; Johann and Voigtländer 2004; Stenger and Voigtländer 2009; Voigtländer 2009b; Christiansen et al. 2010; Bernardy et al. 2010b). It is natural, though, to ask about the quantitative content of free theorems in terms of program efficiency. In a statement like (1), what is the relative performance of the left- and right-hand sides? If we can answer such questions formally,

[*]This author was supported by the DFG under grant VO 1512/1-1.

this will clearly be of particular interest for the mentioned program transformation applications, where statements about efficiency have so far only been made informally or empirically.

In this paper, we lay the ground for formal such investigations. The challenge, of course, as for standard free theorems, is to work independently of concrete function definitions, just as (1) depends on *only the type* of f. To this end, we revise the theory of relational parametricity, essentially marrying it with the classical idea of externalizing the intensional property "computation time" by making it part of the observable program output, and thus accessible to semantic analysis (Wadler 1988; Bjerner and Holmström 1989; Rosendahl 1989; Sands 1995). Our vision is to eventually integrate our results into a tool like `http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi` to enable automatic generation of quantitative free theorems for realistic languages.

To start simple, let us consider some examples. We begin with $f :: \alpha \to \mathsf{Nat}$. The standard free theorem derived from that type is that for every $g :: \tau_1 \to \tau_2$ and $x :: \tau_1$,

$$f\ (g\ x) = f\ x \tag{2}$$

In fact, absent nontermination, it is even possible to conclude that f is a constant function, i.e., for some $n :: \mathsf{Nat}$, f is semantically equivalent to $(\lambda x \to n)$. If we take program runtime into account, then there is another degree of freedom, in addition to picking the natural number n. Namely, two functions of type $\alpha \to \mathsf{Nat}$ can then differ in how long they take before providing their output, because clearly a function that no matter what the input is immediately returns 42 is to be considered different from one that does the same after 7½ million years. Even so, since the same f occurs on the left- and right-hand sides of (2), we can intuitively argue that the right-hand side will never be less efficient than the left-hand side (while it may be more efficient in that it avoids an application of g). On the extensional semantics level, such invariance, namely that different f may use different n in $(\lambda x \to n)$, but the different instantiations of a single polymorphic f at the types τ_2 and τ_1 on the left- and right-hand sides of (2) may not, is exactly what relational parametricity provides. Our task is to formally transfer this argument to the mentioned second degree of freedom, concerning program runtime.

As soon as we do consider runtime, we also have to talk about evaluation order. For the example (2), we can make more precise statements if we know whether function application is call-by-value or call-by-name/need. In the former, strict case, the right-hand side of (2) is actually more efficient than the left-hand side, because the very real cost of applying g is saved. In nonstrict languages, in contrast, the left- and right-hand sides of (2) are to be considered equally efficient since from the type of f we claimed that the function never looks at its argument (extensionally $f = (\lambda x \to n)$ for some arbitrary but fixed n), so the potentially costly inner application $(g\ x)$ on the left-hand side is never actually evaluated. Such issues, and the required reasoning, become more interesting as the types considered get more complicated. For example, for the type $f :: \alpha \to \alpha \to \alpha$ and the associated free theorem

$$f\ (g\ x)\ (g\ y) = g\ (f\ x\ y) \tag{3}$$

the situation is the same as for (2), i.e., the right-hand side is more efficient in a call-by-value language, while no difference is observable with call-by-name/need. But for the type $f :: \alpha \to (\alpha, \alpha)$ and free theorem

$$f\ (g\ x) = \mathit{mapPair}\ (g, g)\ (f\ x) \tag{4}$$

where

$$\begin{aligned} &\mathit{mapPair} :: (\alpha \to \gamma, \beta \to \delta) \to (\alpha, \beta) \to (\gamma, \delta) \\ &\mathit{mapPair}\ (f_1, f_2)\ (x_1, x_2) = (f_1\ x_1, f_2\ x_2) \end{aligned}$$

the situation is rather different: under call-by-value and call-by-need the left-hand side is more efficient, while under call-by-name the left-hand side is for sure not less efficient than the right-hand side, but whether it is actually more efficient depends on what runtime cost we associate with *mapPair*.[1] In summary, the relationships between the runtimes of the various left- and right-hand sides claimed above are as follows:

	$f :: \alpha \to \mathsf{Nat}$	$f :: \alpha \to \alpha \to \alpha$	$f :: \alpha \to (\alpha, \alpha)$
	$f\ (g\ x) = f\ x$	$f\ (g\ x)\ (g\ y) = g\ (f\ x\ y)$	$f\ (g\ x) = \mathit{mapPair}\ (g,g)\ (f\ x)$
call-by-value	lhs $>$ rhs	lhs $>$ rhs	lhs $<$ rhs
call-by-name	lhs $=$ rhs	lhs $=$ rhs	lhs $\leq$ rhs
call-by-need	lhs $=$ rhs	lhs $=$ rhs	lhs $<$ rhs

In this paper we concentrate on call-by-value. From the above, one could jump to the conclusion that then the answer to the question which of the two sides of a free theorem is more efficient depends only on the numbers of syntactic occurrences of g. However, this simplistic view breaks down if one considers types that allow more diverse behavior, like $f :: \alpha \to \alpha \to (\alpha, \alpha)$ or indeed example (1). Also, even for the cases considered above, one should not be deceived by the apparent obviousness of the analysis. For example, that any function $f :: \alpha \to \alpha \to \alpha$ is, by its type alone, not only forced to extensionally be one of the two possible (curried) projections (a fact that can be proved using standard free theorems), but also prevented from causing different costs in different concrete invocations is a nontrivial property that requires proof. To emphasize this point, consider a function $f :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$. Even if we knew that extensionally this function is equivalent to either $(\lambda x\ y \to x)$ or $(\lambda x\ y \to y)$, or even if we knew to which of the two, there would be absolutely no way to conclude which if any of $f\ (g\ x)\ (g\ y)$ and $g\ (f\ x\ y)$ is more efficient for general $g :: \mathsf{Nat} \to \mathsf{Nat}$ and $x, y :: \mathsf{Nat}$.[2] It is only the polymorphism in $f :: \alpha \to \alpha \to \alpha$ that allows such analysis, and what we seek here is the appropriate formal theory as opposed to just the suggestive examples given above.

While the above table may suggest that we are going to prove only comparative statements, actually we will be able to make more precise quantitative statements about the relative costs of left- and right-hand sides of free theorems. For example, for $f :: \alpha \to \alpha \to \alpha$, in the call-by-value setting, we will not only deduce that the left-hand side $f\ (g\ x)\ (g\ y)$ takes more time than the right-hand side $g\ (f\ x\ y)$, but will also obtain that the cost difference is exactly either the cost of applying g on x (without the cost of evaluating x itself) or the cost of applying g on y (without the cost of evaluating y itself).

2 A polymorphically typed lambda-calculus

For formal investigation, we use a relatively small toy language that nevertheless captures essential aspects relevant for our intended analysis. The syntax and typing rules are given in Figures 1 and 2, respectively. There, α ranges over type variables, x, y over term variables, and n over the naturals. The language is explicitly typed, the notation for type annotations is "::", while ":" is the cons operator for lists. The operators **lfold** (corresponding to Haskell's *foldr*) and **ifold** are used to express structural recursion on lists and naturals, respectively. (General, potentially nonterminating, recursion is not included

[1]In principle, one could replace *mapPair* $(g,g)\ (f\ x)$ by **let** $(y_1,y_2) = f\ x$ **in** $(g\ y_1, g\ y_2)$ and consider **let**-binding to be cost-neutral, in which case $f\ (g\ x)$ and the given replacement would be equally efficient under call-by-name. For call-by-value and call-by-need such replacement has no real impact, since for them a whole application of g is saved on the left in any case.

[2]For example, f could be a function that first counts down its first argument to zero, before finally returning its second argument. Then, by choosing g and x appropriately, one could make either of $f\ (g\ x)\ (g\ y)$ and $g\ (f\ x\ y)$ arbitrarily more costly while not affecting the other one at all.

$$\begin{array}{l}\tau ::= \alpha \mid \mathsf{Nat} \mid (\tau,\tau) \mid [\tau] \mid \tau \to \tau \\ t ::= x \mid n \mid \mathbf{case}\ t\ \mathbf{of}\ \{0 \to t\,; x \to t\} \mid t + t \mid [\,]_\tau \mid t : t \mid \mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t\,; x : x \to t\} \mid \\ \qquad (t,t) \mid \mathbf{case}\ t\ \mathbf{of}\ \{(x,x) \to t\} \mid \lambda x :: \tau.t \mid t\ t \mid \mathbf{lfold}(t,t,t) \mid \mathbf{ifold}(t,t,t)\end{array}$$

Figure 1: Syntax of the calculus

$$\Gamma, x :: \tau \vdash x :: \tau \qquad \Gamma \vdash n :: \mathsf{Nat} \qquad \Gamma \vdash [\,]_\tau :: [\tau]$$

$$\frac{\Gamma \vdash t_1 :: \mathsf{Nat} \qquad \Gamma \vdash t_2 :: \mathsf{Nat}}{\Gamma \vdash (t_1 + t_2) :: \mathsf{Nat}} \qquad \frac{\Gamma \vdash t :: \mathsf{Nat} \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma, x :: \mathsf{Nat} \vdash t_2 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1\,; x \to t_2\}) :: \tau}$$

$$\frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]} \qquad \frac{\Gamma \vdash t :: [\tau_1] \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma, x :: \tau_1, y :: [\tau_1] \vdash t_2 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1\,; x : y \to t_2\}) :: \tau}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2) \qquad \Gamma, x :: \tau_1, y :: \tau_2 \vdash t_1 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{(x,y) \to t_1\}) :: \tau}$$

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1.t) :: \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to \tau_2 \to \tau_2 \qquad \Gamma \vdash t_2 :: \tau_2 \qquad \Gamma \vdash t_3 :: [\tau_1]}{\Gamma \vdash \mathbf{lfold}(t_1, t_2, t_3) :: \tau_2}$$

$$\frac{\Gamma \vdash t_1 :: \tau \to \tau \qquad \Gamma \vdash t_2 :: \tau \qquad \Gamma \vdash t_3 :: \mathsf{Nat}}{\Gamma \vdash \mathbf{ifold}(t_1, t_2, t_3) :: \tau}$$

Figure 2: Typing rules

for simplicity.) For example, the function *mapList* from the introduction is defined in our calculus as follows:

$$\mathit{mapList} = \lambda g :: (\alpha \to \beta).\lambda ys :: [\alpha].\mathbf{lfold}(\lambda x :: \alpha.\lambda xs :: [\beta].(g\ x) : xs, [\,]_\beta, ys)$$

and satisfies $\alpha, \beta \vdash \mathit{mapList} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$.

Semantically, types are interpreted as sets in an absolutely standard way, see Figure 3 (where θ is a mapping from type variables to sets). There is also a standard denotational term semantics, shown in Figure 4, which satisfies: if $\Gamma \vdash t :: \tau$, then $[\![t]\!]_\sigma \in [\![\tau]\!]_\theta$ for every σ with $\sigma(x) \in [\![\tau']\!]_\theta$ for every $x :: \tau'$ in Γ.

The key to relational parametricity, and thus to free theorems, is to provide a suitable interpretation of types as relations. The standard such type-indexed family of relations for our setting so far, defined by induction on the structure of types, and called a "logical relation", is given in Figure 5 (where ρ is a mapping from type variables to binary relations between sets). Note that we use juxtaposition ($\mathbf{f}\ \mathbf{x}$), instead of $\mathbf{f}(\mathbf{x})$, as notation for applying mathematical functions (mirroring the syntactic application on term level). Also, we use the following definitions:

$$\begin{aligned} \mathit{lift}_{[\,]}(R) &= \{([\mathbf{x_1}, \ldots, \mathbf{x_n}], [\mathbf{y_1}, \ldots, \mathbf{y_n}]) \mid n \in \mathbb{N} \wedge \forall i \in \{1, \ldots, n\}.\ (\mathbf{x_i}, \mathbf{y_i}) \in R\} \\ \mathit{lift}_{(,)}(R_1, R_2) &= \{((\mathbf{x_1}, \mathbf{x_2}), (\mathbf{y_1}, \mathbf{y_2})) \mid (\mathbf{x_1}, \mathbf{y_1}) \in R_1 \wedge (\mathbf{x_2}, \mathbf{y_2}) \in R_2\} \end{aligned}$$

$$
\begin{aligned}
[\![\alpha]\!]_\theta &= \theta(\alpha) && \text{(an arbitrary set, fixed in } \theta\text{)}\\
[\![\mathsf{Nat}]\!]_\theta &= \mathbb{N} && \text{(the naturals)}\\
[\![[\tau]]\!]_\theta &= \{[\mathbf{x_1},\ldots,\mathbf{x_n}] \mid n \in \mathbb{N} \wedge \forall i \in \{1,\ldots,n\}.\ \mathbf{x_i} \in [\![\tau]\!]_\theta\} && \text{(the free monoid over a set)}\\
[\![(\tau_1,\tau_2)]\!]_\theta &= [\![\tau_1]\!]_\theta \times [\![\tau_2]\!]_\theta && \text{(the Cartesian product of sets)}\\
[\![\tau_1 \to \tau_2]\!]_\theta &= [\![\tau_2]\!]_\theta{}^{[\![\tau_1]\!]_\theta} && \text{(the mathematical function space between sets)}
\end{aligned}
$$

Figure 3: Standard type semantics

$$
\begin{aligned}
[\![x]\!]_\sigma &= \sigma(x)\\
[\![n]\!]_\sigma &= \mathbf{n}\\
[\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1 ; x \to t_2\}]\!]_\sigma &= \begin{cases} [\![t_1]\!]_\sigma & \text{if } [\![t]\!]_\sigma = \mathbf{0}\\ [\![t_2]\!]_{\sigma[x\mapsto\mathbf{n}]} & \text{if } [\![t]\!]_\sigma = \mathbf{n},\ \mathbf{n} > \mathbf{0}\end{cases}\\
[\![t_1 + t_2]\!]_\sigma &= [\![t_1]\!]_\sigma + [\![t_2]\!]_\sigma\\
[\![[\,]_\tau]\!]_\sigma &= [\,]\\
[\![t_1 : t_2]\!]_\sigma &= [[\![t_1]\!]_\sigma, \mathbf{v_1},\ldots,\mathbf{v_n}] \text{ with } [\![t_2]\!]_\sigma = [\mathbf{v_1},\ldots,\mathbf{v_n}]\\
[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1 ; x : y \to t_2\}]\!]_\sigma &= \begin{cases} [\![t_1]\!]_\sigma & \text{if } [\![t]\!]_\sigma = [\,]\\ [\![t_2]\!]_{\sigma[x\mapsto\mathbf{v_1}, y\mapsto[\mathbf{v_2},\ldots,\mathbf{v_n}]]} & \text{if } [\![t]\!]_\sigma = [\mathbf{v_1},\ldots,\mathbf{v_n}],\ \mathbf{n} > \mathbf{0}\end{cases}\\
[\![(t_1,t_2)]\!]_\sigma &= ([\![t_1]\!]_\sigma, [\![t_2]\!]_\sigma)\\
[\![\mathbf{case}\ t\ \mathbf{of}\ \{(x,y) \to t_1\}]\!]_\sigma &= [\![t_1]\!]_{\sigma[x\mapsto\mathbf{v_1}, y\mapsto\mathbf{v_2}]} \text{ with } [\![t]\!]_\sigma = (\mathbf{v_1},\mathbf{v_2})\\
[\![\lambda x :: \tau.t]\!]_\sigma &= \lambda\mathbf{v}.[\![t]\!]_{\sigma[x\mapsto\mathbf{v}]}\\
[\![t_1\ t_2]\!]_\sigma &= [\![t_1]\!]_\sigma\ [\![t_2]\!]_\sigma\\
[\![\mathbf{lfold}(t_1,t_2,t_3)]\!]_\sigma &= [\![t_1]\!]_\sigma\ \mathbf{v_1}\ ([\![t_1]\!]_\sigma\ \mathbf{v_2}\ \ldots([\![t_1]\!]_\sigma\ \mathbf{v_n}\ [\![t_2]\!]_\sigma)\ldots) \text{ with } [\![t_3]\!]_\sigma = [\mathbf{v_1},\ldots,\mathbf{v_n}]\\
[\![\mathbf{ifold}(t_1,t_2,t_3)]\!]_\sigma &= \underbrace{[\![t_1]\!]_\sigma\ ([\![t_1]\!]_\sigma\ \ldots([\![t_1]\!]_\sigma}_{[\![t_3]\!]_\sigma \text{ times}}\ [\![t_2]\!]_\sigma)\ldots)
\end{aligned}
$$

Figure 4: Standard term semantics

$$
\begin{aligned}
\Delta_{\alpha,\rho} &= \rho(\alpha)\\
\Delta_{\mathsf{Nat},\rho} &= id_{\mathbb{N}}\\
\Delta_{[\tau],\rho} &= \mathit{lift}_{[\,]}(\Delta_{\tau,\rho})\\
\Delta_{(\tau_1,\tau_2),\rho} &= \mathit{lift}_{(,)}(\Delta_{\tau_1,\rho}, \Delta_{\tau_2,\rho})\\
\Delta_{\tau_1\to\tau_2,\rho} &= \{(\mathbf{f},\mathbf{g}) \mid \forall(\mathbf{x},\mathbf{y}) \in \Delta_{\tau_1,\rho}.\ (\mathbf{f}\ \mathbf{x}, \mathbf{g}\ \mathbf{y}) \in \Delta_{\tau_2,\rho}\}
\end{aligned}
$$

Figure 5: Standard logical relation

To derive free theorems, all one needs is the following theorem (Reynolds 1983; Wadler 1989). In it, *Rel* denotes the collection of all binary relations between sets. (Later, we also use $Rel(S_1,S_2)$ to denote more specifically the collection of all binary relations between sets S_1 and S_2.)

Theorem 1 (standard parametricity theorem)**.** *If* $\Gamma \vdash t :: \tau$*, then for every* ρ*,* σ_1*,* σ_2 *such that*

- *for every* α *in* Γ*,* $\rho(\alpha) \in Rel$*, and*
- *for every* $x :: \tau'$ *in* Γ*,* $(\sigma_1(x),\sigma_2(x)) \in \Delta_{\tau',\rho}$ *,*

we have $([\![t]\!]_{\sigma_1},[\![t]\!]_{\sigma_2}) \in \Delta_{\tau,\rho}$.

Our aim now is to provide an analogous theorem for a setting in which computation costs are taken into account. For doing so, we clearly first need to develop the underlying semantic notions (and then a suitable new logical relation).

3 Adding costs to the semantics

As already mentioned in the introduction, we want to study the call-by-value case here. That is, we consider the presented lambda-calculus as a small core language of a kind of strict Haskell or pure ML.

In order to reflect computation costs in the semantics, we first revise the set interpretation of types. In addition to a value, every semantic object now has to carry an integer representing some abstract notion of costs incurred while computing that value. Such integers (actually naturals would suffice for the moment, but the added generality of negative numbers comes in handy later on) need to be added only at top-level positions of compound values, thanks to our restriction to strict evaluation. For example, the costs of individual list elements are not relevant ultimately, only the cost of a whole list, because anyway there is no means to evaluate only a part of it (as there would be in a nonstrict language). The only place where "embedded" costs are relevant is in (the result positions of) function spaces, because there it is really important to capture which actual function arguments lead to which specific costs in the output. Formally, we define a variant of the mapping from Figure 3 in Figure 6, where $\mathscr{C}(S) = \{(\mathbf{x},c) \mid \mathbf{x} \in S \wedge c \in \mathbb{Z}\}$. That new mapping, $[\![\cdot]\!]'$, does not itself capture top-level costs. But ultimately, instead of the earlier $[\![t]\!]_\sigma \in [\![\tau]\!]_\theta$ we will have that a term t of type τ is mapped, by a new term semantics, to an element of the $\mathscr{C}(\cdot)$*-lifting* of $[\![\tau]\!]'_\theta$.

Our new term semantics (changed from Figure 4) follows the same spirit as the instrumented semantics of Rosendahl (1989). Essentially, the cost integers are carried around and just suitably propagated, except where we decide that a certain semantic operation should be counted as contributing a cost of its own. Here we assign a cost only to the invocation of functions, so we add a cost of 1 in the interpretation of lambda-abstractions.[3] The formal definition is given in Figure 7. The helper function $\triangleright$ defined in the figure adds, in $c \triangleright \mathbf{x}$, the cost c to the cost component of semantic object $\mathbf{x}$. The other helper functions are cost-propagating versions of data constructors and function application. Syntactically, $\triangleright$ and $:^{¢}$ are right-associative, $¢$ is left-associative, and $\triangleright$ has higher precedence than the other semantic operations. Now we have that if $\Gamma \vdash t :: \tau$ then $[\![t]\!]^{¢}_\sigma \in \mathscr{C}([\![\tau]\!]'_\theta)$ for every θ mapping the type variables in Γ to sets and σ with $\sigma(x) \in [\![\tau']\!]'_\theta$ for every $x :: \tau'$ in Γ.

Example 1. Let $length = \lambda xs :: [\alpha].\mathbf{lfold}(\lambda x :: \alpha.\lambda y :: \mathsf{Nat}.1+y,0,xs)$. We calculate the semantics of $length[\mathsf{Nat}/\alpha]\ (1:2:[]_{\mathsf{Nat}})$, where $[\mathsf{Nat}/\alpha]$ denotes syntactic substitution of Nat for all occurrences of

[3] Other possible places to put extra costs would have been the data constructors and **case**-expressions. Actually, we have found that our general results, in particular Theorem 2, are unaffected by such changes.

$$\begin{aligned}
[\![\alpha]\!]'_\theta &= \theta(\alpha)\\
[\![\mathsf{Nat}]\!]'_\theta &= \mathbb{N}\\
[\![[\tau]]\!]'_\theta &= \{[\mathbf{x_1},\ldots,\mathbf{x_n}] \mid n \in \mathbb{N} \wedge \forall i \in \{1,\ldots,n\}.\ \mathbf{x_i} \in [\![\tau]\!]'_\theta\}\\
[\![(\tau_1,\tau_2)]\!]'_\theta &= [\![\tau_1]\!]'_\theta \times [\![\tau_2]\!]'_\theta\\
[\![\tau_1 \to \tau_2]\!]'_\theta &= \mathscr{C}([\![\tau_2]\!]'_\theta)^{[\![\tau_1]\!]'_\theta}
\end{aligned}$$

Figure 6: Type semantics with embedded costs

$$\begin{aligned}
[\![x]\!]^{¢}_\sigma &= (\sigma(x),0)\\
[\![n]\!]^{¢}_\sigma &= (\mathbf{n},0)\\
[\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1\,; x \to t_2\}]\!]^{¢}_\sigma &= \begin{cases} c \triangleright [\![t_1]\!]^{¢}_\sigma & \text{if } [\![t]\!]^{¢}_\sigma = (\mathbf{0},c)\\ c \triangleright [\![t_2]\!]^{¢}_{\sigma[x\mapsto\mathbf{n}]} & \text{if } [\![t]\!]^{¢}_\sigma = (\mathbf{n},c),\ \mathbf{n} > \mathbf{0}\end{cases}\\
[\![t_1 + t_2]\!]^{¢}_\sigma &= (\mathbf{n_1}+\mathbf{n_2}, c_1+c_2)\ \text{ with } [\![t_1]\!]^{¢}_\sigma = (\mathbf{n_1},c_1),\ [\![t_2]\!]^{¢}_\sigma = (\mathbf{n_2},c_2)\\
[\![[\,]_\tau]\!]^{¢}_\sigma &= ([\,],0)\\
[\![t_1 : t_2]\!]^{¢}_\sigma &= [\![t_1]\!]^{¢}_\sigma :^{¢} [\![t_2]\!]^{¢}_\sigma\\
[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1\,; x : y \to t_2\}]\!]^{¢}_\sigma &= \begin{cases} c \triangleright [\![t_1]\!]^{¢}_\sigma & \text{if } [\![t]\!]^{¢}_\sigma = ([\,],c)\\ c \triangleright [\![t_2]\!]^{¢}_{\sigma[x\mapsto\mathbf{v_1},y\mapsto[\mathbf{v_2},\ldots,\mathbf{v_n}]]} & \text{if } [\![t]\!]^{¢}_\sigma = ([\mathbf{v_1},\ldots,\mathbf{v_n}],c),\ \mathbf{n} > \mathbf{0}\end{cases}\\
[\![(t_1,t_2)]\!]^{¢}_\sigma &= ([\![t_1]\!]^{¢}_\sigma, [\![t_2]\!]^{¢}_\sigma)^{¢}\\
[\![\mathbf{case}\ t\ \mathbf{of}\ \{(x,y) \to t_1\}]\!]^{¢}_\sigma &= c \triangleright [\![t_1]\!]^{¢}_{\sigma[x\mapsto\mathbf{v_1},y\mapsto\mathbf{v_2}]}\ \text{ with } [\![t]\!]^{¢}_\sigma = ((\mathbf{v_1},\mathbf{v_2}),c)\\
[\![\lambda x :: \tau.t]\!]^{¢}_\sigma &= (\lambda\mathbf{v}.1 \triangleright [\![t]\!]^{¢}_{\sigma[x\mapsto\mathbf{v}]},0)\\
[\![t_1\ t_2]\!]^{¢}_\sigma &= [\![t_1]\!]^{¢}_\sigma\ ¢\ [\![t_2]\!]^{¢}_\sigma\\
[\![\mathbf{lfold}(t_1,t_2,t_3)]\!]^{¢}_\sigma &= (c_1+c_3) \triangleright ((\mathbf{g}\ \mathbf{v_1})\ ¢\ ((\mathbf{g}\ \mathbf{v_2})\ ¢\ \ldots((\mathbf{g}\ \mathbf{v_n})\ ¢\ [\![t_2]\!]^{¢}_\sigma)\ldots))\\
&\ \text{with } [\![t_1]\!]^{¢}_\sigma = (\mathbf{g},c_1),\ [\![t_3]\!]^{¢}_\sigma = ([\mathbf{v_1},\ldots,\mathbf{v_n}],c_3)\\
[\![\mathbf{ifold}(t_1,t_2,t_3)]\!]^{¢}_\sigma &= (c_1+c_3) \triangleright (\underbrace{(\mathbf{g},0)\ ¢\ ((\mathbf{g},0)\ ¢\ \ldots((\mathbf{g},0)\ ¢}_{\mathbf{n}\text{ times}} [\![t_2]\!]^{¢}_\sigma)\ldots))\\
&\ \text{with } [\![t_1]\!]^{¢}_\sigma = (\mathbf{g},c_1),\ [\![t_3]\!]^{¢}_\sigma = (\mathbf{n},c_3)
\end{aligned}$$

where

$$\begin{aligned}
c \triangleright (\mathbf{v},c') &= (\mathbf{v}, c+c')\\
\mathbf{x} :^{¢} \mathbf{xs} &= ([\mathbf{v},\mathbf{v_1},\ldots,\mathbf{v_n}], c+c')\ \text{ with } \mathbf{x} = (\mathbf{v},c),\ \mathbf{xs} = ([\mathbf{v_1},\ldots,\mathbf{v_n}],c')\\
(\mathbf{x_1},\mathbf{x_2})^{¢} &= ((\mathbf{v_1},\mathbf{v_2}), c+c')\ \text{ with } \mathbf{x_1} = (\mathbf{v_1},c),\ \mathbf{x_2} = (\mathbf{v_2},c')\\
\mathbf{f}\ ¢\ \mathbf{x} &= (c+c') \triangleright (\mathbf{g}\ \mathbf{v})\ \text{ with } \mathbf{f} = (\mathbf{g},c),\ \mathbf{x} = (\mathbf{v},c')
\end{aligned}$$

Figure 7: Term semantics with costs

$$\begin{aligned}
\Delta'_{\alpha,\rho} &= \rho(\alpha)\\
\Delta'_{\mathsf{Nat},\rho} &= id_{\mathbb{N}}\\
\Delta'_{[\tau],\rho} &= \mathit{lift}_{[]}(\Delta'_{\tau,\rho})\\
\Delta'_{(\tau_1,\tau_2),\rho} &= \mathit{lift}_{(,)}(\Delta'_{\tau_1,\rho},\Delta'_{\tau_2,\rho})\\
\Delta'_{\tau_1\to\tau_2,\rho} &= \{(\mathbf{f},\mathbf{g}) \mid \forall(\mathbf{x},\mathbf{y})\in\Delta'_{\tau_1,\rho}.\ (\mathbf{f}\ \mathbf{x},\mathbf{g}\ \mathbf{y})\in\mathscr{C}(\Delta'_{\tau_2,\rho})\}
\end{aligned}$$

Figure 8: Logical relation with embedded costs

α, as follows:

$$\begin{aligned}
&[\![(\lambda xs :: [\mathsf{Nat}].\mathbf{lfold}(\lambda x :: \mathsf{Nat}.\lambda y :: \mathsf{Nat}.1+y,0,xs))\ (1:2:[]_{\mathsf{Nat}})]\!]^{¢}_{\emptyset}\\
&= (\lambda\mathbf{v}.1 \rhd [\![\mathbf{lfold}(\lambda x :: \mathsf{Nat}.\lambda y :: \mathsf{Nat}.1+y,0,xs)]\!]^{¢}_{[xs\mapsto\mathbf{v}]},0)\ ¢\ ([\mathbf{1},\mathbf{2}],0)\\
&= 1 \rhd [\![\mathbf{lfold}(\lambda x :: \mathsf{Nat}.\lambda y :: \mathsf{Nat}.1+y,0,xs)]\!]^{¢}_{[xs\mapsto[\mathbf{1},\mathbf{2}]]}\\
&= 1 \rhd (((\lambda\mathbf{x}.(\lambda\mathbf{y}.(\mathbf{1}+\mathbf{y},1),1))\ \mathbf{1})\ ¢\ (((\lambda\mathbf{x}.(\lambda\mathbf{y}.(\mathbf{1}+\mathbf{y},1),1))\ \mathbf{2})\ ¢\ (\mathbf{0},0)))\\
&= 1 \rhd ((\lambda\mathbf{y}.(\mathbf{1}+\mathbf{y},1),1)\ ¢\ 1 \rhd (\mathbf{1}+\mathbf{0},1))\\
&= 1 \rhd (1+1+1) \rhd ((\lambda\mathbf{y}.(\mathbf{1}+\mathbf{y},1))\ (\mathbf{1}+\mathbf{0}))\\
&= (\mathbf{2},5)
\end{aligned}$$

Exactly the five required beta-reductions (once for $\lambda xs :: [\mathsf{Nat}]$ and twice each for $\lambda x :: \mathsf{Nat}.\lambda y :: \mathsf{Nat}$) have been counted.

Note that due to the way we handle polymorphism, a $[\![t]\!]^{¢}_{\sigma}$ can be element of $\mathscr{C}([\![\tau]\!]'_{\theta_1})$ and $\mathscr{C}([\![\tau]\!]'_{\theta_2})$ for completely different θ_1 and θ_2. For example, $[\![(\lambda x :: \alpha.x)]\!]^{¢}_{\emptyset}$ is $(\mathbf{g},0)$ where $\mathbf{g}$ maps $\mathbf{v}\in S$ to $(\mathbf{v},1)\in\mathscr{C}(S)$, for *every* set S. (We denote by $\emptyset$ an empty mapping.)

Lemma 1. *Let $\Gamma \vdash t :: \tau$, where Γ contains no term variables. For every type variable α, type τ' not containing type variables, and θ mapping the type variables in $\Gamma\setminus\{\alpha\}$ to sets, we have $[\![t[\tau'/\alpha]]\!]^{¢}_{\emptyset} \in \mathscr{C}([\![\tau[\tau'/\alpha]]\!]'_{\theta})$. Moreover, $[\![\tau[\tau'/\alpha]]\!]'_{\theta} = [\![\tau]\!]'_{\theta[\alpha\mapsto[\![\tau']\!]'_{\emptyset}]}$, and while $[\![t]\!]^{¢}_{\emptyset}$ is an element of $\mathscr{C}([\![\tau]\!]'_{\theta[\alpha\mapsto S]})$ for* arbitrary *S, for the* specific *case $S = [\![\tau']\!]'_{\emptyset}$ we have $[\![t]\!]^{¢}_{\emptyset} = [\![t[\tau'/\alpha]]\!]^{¢}_{\emptyset}$.*

We also note some simple properties of the semantic operations; these properties will henceforth be used freely without explicit mention:

- $c \rhd c' \rhd \mathbf{x} = (c+c') \rhd \mathbf{x}$
- $c \rhd (\mathbf{x}_1,\mathbf{x}_2)^{¢} = (c \rhd \mathbf{x}_1,\mathbf{x}_2)^{¢} = (\mathbf{x}_1, c \rhd \mathbf{x}_2)^{¢}$
- $c \rhd (\mathbf{x} :^{¢} \mathbf{xs}) = c \rhd \mathbf{x} :^{¢} \mathbf{xs} = \mathbf{x} :^{¢} c \rhd \mathbf{xs}$
- $c \rhd (\mathbf{f}\ ¢\ \mathbf{x}) = c \rhd \mathbf{f}\ ¢\ \mathbf{x} = \mathbf{f}\ ¢\ c \rhd \mathbf{x}$

4 New relational interpretations of types

Now we also need a new interpretation of types as relations, i.e., a new logical relation. We get directions by comparing the set interpretations from Figures 3 and 6. There, a difference only appears for the output side of function arrows, namely the codomain is lifted to a costful setting. We try the same on the relational level and thus transform the logical relation from Figure 5 into the one given in Figure 8, where $\mathscr{C}(R) = \{((\mathbf{x},c),(\mathbf{y},c)) \mid (\mathbf{x},\mathbf{y})\in R \wedge c\in\mathbb{Z}\}$.

$$
\begin{aligned}
\Delta^{¢}_{\alpha,\rho} &= \mathscr{C}(\rho(\alpha)) \\
\Delta^{¢}_{\mathsf{Nat},\rho} &= id_{\mathscr{C}(\mathbb{N})} \\
\Delta^{¢}_{[\tau],\rho} &= \mathit{lift}^{¢}_{[]}(\Delta^{¢}_{\tau,\rho}) \\
\Delta^{¢}_{(\tau_1,\tau_2),\rho} &= \mathit{lift}^{¢}_{(,)}(\Delta^{¢}_{\tau_1,\rho},\Delta^{¢}_{\tau_2,\rho}) \\
\Delta^{¢}_{\tau_1\to\tau_2,\rho} &= \{(\mathbf{f},\mathbf{g}) \mid \mathit{cost}(\mathbf{f}) = \mathit{cost}(\mathbf{g}) \wedge \forall(\mathbf{x},\mathbf{y}) \in \Delta^{¢}_{\tau_1,\rho}.\ (\mathbf{f} ¢\ \mathbf{x}, \mathbf{g} ¢\ \mathbf{y}) \in \Delta^{¢}_{\tau_2,\rho}\}
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{lift}^{¢}_{[]}(R^{¢}) &= \{([\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢},[\mathbf{y_1},\ldots,\mathbf{y_n}]^{¢}) \mid n \in \mathbb{N} \wedge \forall i \in \{1,\ldots,n\}.\ (\mathbf{x_i},\mathbf{y_i}) \in R^{¢}\} \\
\mathit{lift}^{¢}_{(,)}(R^{¢}_1,R^{¢}_2) &= \{((\mathbf{x_1},\mathbf{x_2})^{¢},(\mathbf{y_1},\mathbf{y_2})^{¢}) \mid (\mathbf{x_1},\mathbf{y_1}) \in R^{¢}_1 \wedge (\mathbf{x_2},\mathbf{y_2}) \in R^{¢}_2\}
\end{aligned}
$$

and $[\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢}$ abbreviates $\mathbf{x_1} :^{¢} \ldots :^{¢} \mathbf{x_n} :^{¢} ([],0)$.

Figure 9: Fully cost-lifted logical relation

Note that ρ in Figure 8 still maps to "normal" binary relations between sets, rather than to $\mathscr{C}(\cdot)$-lifted ones. In turn, the $[\![\cdot]\!]^{¢}$-semantics of terms will be related by the $\mathscr{C}(\cdot)$*-lifting* of Δ'. Indeed, a proof very similar to that of Theorem 1, by induction on typing derivations, establishes the following theorem. (The proof is sketched in Appendix A.)

Theorem 2. *If* $\Gamma \vdash t :: \tau$*, then for every* ρ*,* σ_1*,* σ_2 *such that*

- *for every* α *in* Γ*,* $\rho(\alpha) \in \mathit{Rel}$*, and*
- *for every* $x :: \tau'$ *in* Γ*,* $(\sigma_1(x),\sigma_2(x)) \in \Delta'_{\tau',\rho}$ *,*

we have $([\![t]\!]^{¢}_{\sigma_1},[\![t]\!]^{¢}_{\sigma_2}) \in \mathscr{C}(\Delta'_{\tau,\rho})$*.*

One of the key cases in the proof, for function application, uses that $(\mathbf{f},\mathbf{g}) \in \mathscr{C}(\Delta'_{\tau_1\to\tau_2,\rho})$ implies $\forall(\mathbf{x},\mathbf{y}) \in \mathscr{C}(\Delta'_{\tau_1,\rho}).\ (\mathbf{f} ¢\ \mathbf{x}, \mathbf{g} ¢\ \mathbf{y}) \in \mathscr{C}(\Delta'_{\tau_2,\rho})$. Note the subtle differences here to the definition of $\Delta'_{\tau_1\to\tau_2,\rho}$ in Figure 8, namely the $\mathscr{C}(\cdot)$-lifting on both $\Delta'_{\tau_1\to\tau_2,\rho}$ and $\Delta'_{\tau_1,\rho}$, and hence the use of $(\mathbf{f} ¢\ \mathbf{x}, \mathbf{g} ¢\ \mathbf{y})$ instead of $(\mathbf{f}\ \mathbf{x}, \mathbf{g}\ \mathbf{y})$. Working fully on the $\mathscr{C}(\cdot)$-lifted level is also preferable in later derivations of free theorems (based on the logical relation), so it seems a good idea to provide an alternative definition of relational interpretations of types that does not mix unlifted (like $\Delta'_{\tau_1,\rho}$) and lifted (like $\mathscr{C}(\Delta'_{\tau_2,\rho})$) uses. However, we have to be careful, because the "implies" in the first sentence of the current paragraph is really just that: an implication, not an equivalence. In order to give a direct inductive definition for $\mathscr{C}(\Delta'_{\cdot,\cdot})$, we need exact characterizations. For the case of function types, the following lemma is easily obtained from the definitions, where, in general, $\mathit{cost}((\mathbf{v},c)) = c$.

Lemma 2. $(\mathbf{f},\mathbf{g}) \in \mathscr{C}(\Delta'_{\tau_1\to\tau_2,\rho}) \Leftrightarrow \mathit{cost}(\mathbf{f}) = \mathit{cost}(\mathbf{g}) \wedge \forall(\mathbf{x},\mathbf{y}) \in \mathscr{C}(\Delta'_{\tau_1,\rho}).\ (\mathbf{f} ¢\ \mathbf{x}, \mathbf{g} ¢\ \mathbf{y}) \in \mathscr{C}(\Delta'_{\tau_2,\rho})$

Using similar characterizations for the other cases, we arrive at the new logical relation given in Figure 9, which is connected to the one from Figure 8 by the following (inductively proved) lemma.

Lemma 3. *For every* τ *and* ρ*,* $\mathscr{C}(\Delta'_{\tau,\rho}) = \Delta^{¢}_{\tau,\rho}$*.*

Together with Theorem 2, we immediately get:

Corollary 1. *If* $\Gamma \vdash t :: \tau$*, then for every* ρ*,* σ_1*,* σ_2 *such that*

- *for every* α *in* Γ*,* $\rho(\alpha) \in \mathit{Rel}$*, and*
- *for every* $x :: \tau'$ *in* Γ*,* $((\sigma_1(x),0),(\sigma_2(x),0)) \in \Delta^{¢}_{\tau',\rho}$ *,*

we have $([\![t]\!]^{¢}_{\sigma_1},[\![t]\!]^{¢}_{\sigma_2}) \in \Delta^{¢}_{\tau,\rho}$*.*

5 Deriving free theorems

Now we can go for applications of Corollary 1 to specific polymorphic types, in order to derive cost-aware statements about terms of those types. First, we need some auxiliary notions. In addition to $cost((\mathbf{v},c)) = c$ we define $val((\mathbf{v},c)) = \mathbf{v}$, and for every $\mathbf{f} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1})$ and $\mathbf{x} \in \mathscr{C}(S_1)$, for some sets S_1 and S_2, $appCost(\mathbf{f},\mathbf{x}) = cost(\mathbf{f} \cent \mathbf{x}) - cost(\mathbf{x})$. Also, a standard way of deriving free theorems is to specialize relations (those mapped to by ρ) to the graphs of functions. In our setting, we have to be careful to get the "$\mathscr{C}(\cdot)$-lifting level" right. Moreover, since in our derivations of free theorems we will need to have access to information about the costs associated to specific function arguments and results, it is helpful to make specialized relations as tightly specified as possible. Hence, instead of the full function graphs commonly used, we will go for finite parts thereof. So given sets S_1 and S_2, a $\mathscr{C}(\cdot)$-lifted function $\mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1})$, and $\mathscr{C}(\cdot)$-lifted values $\mathbf{x_1},\ldots,\mathbf{x_n} \in \mathscr{C}(S_1)$, with $n \in \mathbb{N}$, we define:

$$R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}} = \{(val(\mathbf{x_1}), val(\mathbf{g} \cent \mathbf{x_1})),\ldots,(val(\mathbf{x_n}), val(\mathbf{g} \cent \mathbf{x_n}))\} \in Rel(S_1,S_2)$$

The crucial property, directly derived from definitions, (and a simple corollary of it) we are going to exploit about $R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}$ can be given as follows (under the given conditions on S_1, S_2, $\mathbf{g}$, and $\mathbf{x_1},\ldots,\mathbf{x_n}$):

Proposition 1. *Let $\mathbf{x} \in \mathscr{C}(S_1)$ and $\mathbf{y} \in \mathscr{C}(S_2)$. Then $(\mathbf{x},\mathbf{y}) \in \mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}})$ if and only if there exist $i \in \{1,\ldots,n\}$ and $c \in \mathbb{Z}$ such that $\mathbf{x} = c \rhd appCost(\mathbf{g},\mathbf{x_i}) \rhd \mathbf{x_i}$ and $\mathbf{y} = c \rhd (\mathbf{g} \cent \mathbf{x_i})$.*

Corollary 2. *Let $\mathbf{x} \in \mathscr{C}(S_1)$ and $\mathbf{y} \in \mathscr{C}(S_2)$. If $(\mathbf{x},\mathbf{y}) \in \mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}})$, then there exists $i \in \{1,\ldots,n\}$ such that $\mathbf{g} \cent \mathbf{x} = appCost(\mathbf{g},\mathbf{x_i}) \rhd \mathbf{y}$.*

Let us now derive a first concrete free (improvement) theorem, for one of the types from Section 1.

Example 2. Let some term f be given with $\alpha \vdash f :: \alpha \to \alpha \to \alpha$. By Corollary 1 we have:

$$\forall R \in Rel.\ ([\![f]\!]^{\cent}_{\emptyset}, [\![f]\!]^{\cent}_{\emptyset}) \in \Delta^{\cent}_{\alpha\to\alpha\to\alpha,[\alpha\mapsto R]}$$

By the definition of the logical relation in Figure 9 this implies:

$$\forall R \in Rel, (\mathbf{x},\mathbf{y}),(\mathbf{x}',\mathbf{y}') \in \mathscr{C}(R).\ ([\![f]\!]^{\cent}_{\emptyset} \cent \mathbf{x} \cent \mathbf{x}', [\![f]\!]^{\cent}_{\emptyset} \cent \mathbf{y} \cent \mathbf{y}') \in \mathscr{C}(R)$$

Specialization of R gives:

$$\begin{array}{l}\forall S_1,S_2 \text{ sets}, \mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1}), \mathbf{x_1},\mathbf{x_2} \in \mathscr{C}(S_1).\\ \ \forall (\mathbf{x},\mathbf{y}),(\mathbf{x}',\mathbf{y}') \in \mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1},\mathbf{x_2}}).\ ([\![f]\!]^{\cent}_{\emptyset} \cent \mathbf{x} \cent \mathbf{x}', [\![f]\!]^{\cent}_{\emptyset} \cent \mathbf{y} \cent \mathbf{y}') \in \mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1},\mathbf{x_2}})\end{array}$$

From this follows, by Proposition 1:

$$\begin{array}{l}\forall S_1,S_2 \text{ sets}, \mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1}), \mathbf{x_1},\mathbf{x_2} \in \mathscr{C}(S_1).\\ ([\![f]\!]^{\cent}_{\emptyset} \cent (appCost(\mathbf{g},\mathbf{x_1}) \rhd \mathbf{x_1}) \cent (appCost(\mathbf{g},\mathbf{x_2}) \rhd \mathbf{x_2}), [\![f]\!]^{\cent}_{\emptyset} \cent (\mathbf{g} \cent \mathbf{x_1}) \cent (\mathbf{g} \cent \mathbf{x_2})) \in \mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1},\mathbf{x_2}})\end{array}$$

which in turn implies, by Corollary 2:

$$\begin{array}{l}\forall S_1,S_2 \text{ sets}, \mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1}), \mathbf{x_1},\mathbf{x_2} \in \mathscr{C}(S_1).\ \exists i \in \{1,2\}.\\ \mathbf{g} \cent ([\![f]\!]^{\cent}_{\emptyset} \cent (appCost(\mathbf{g},\mathbf{x_1}) \rhd \mathbf{x_1}) \cent (appCost(\mathbf{g},\mathbf{x_2}) \rhd \mathbf{x_2}))\\ = appCost(\mathbf{g},\mathbf{x_i}) \rhd ([\![f]\!]^{\cent}_{\emptyset} \cent (\mathbf{g} \cent \mathbf{x_1}) \cent (\mathbf{g} \cent \mathbf{x_2}))\end{array}$$

which simplifies to:

$$\begin{array}{l}\forall S_1,S_2 \text{ sets}, \mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1}), \mathbf{x_1},\mathbf{x_2} \in \mathscr{C}(S_1).\ \exists c \in \{appCost(\mathbf{g},\mathbf{x_1}), appCost(\mathbf{g},\mathbf{x_2})\}.\\ c \rhd (\mathbf{g} \cent ([\![f]\!]^{\cent}_{\emptyset} \cent \mathbf{x_1} \cent \mathbf{x_2})) = [\![f]\!]^{\cent}_{\emptyset} \cent (\mathbf{g} \cent \mathbf{x_1}) \cent (\mathbf{g} \cent \mathbf{x_2})\end{array}$$

By using the definitions from Figures 6 and 7, and Lemma 1, we can conclude that:

$$\begin{array}{l}\forall \tau_1,\tau_2 \text{ types}, g :: \tau_1 \to \tau_2, t_1 :: \tau_1, t_2 :: \tau_1.\ \exists c \in \{\mathit{appCost}([\![g]\!]^{¢}_{\emptyset},[\![t_1]\!]^{¢}_{\emptyset}), \mathit{appCost}([\![g]\!]^{¢}_{\emptyset},[\![t_2]\!]^{¢}_{\emptyset})\}.\\ \quad c \rhd [\![g\ (f[\tau_1/\alpha]\ t_1\ t_2)]\!]^{¢}_{\emptyset} = [\![f[\tau_2/\alpha]\ (g\ t_1)\ (g\ t_2)]\!]^{¢}_{\emptyset}\end{array}$$

This certainly means that the right-hand side of (3) in the introduction is more efficient than its left-hand side. Indeed, after defining "$\mathbf{v} \sqsubseteq \mathbf{v}'$" as "$\exists c \geq 0.\ c \rhd \mathbf{v} = \mathbf{v}'$" (or, equivalently, "$\mathit{val}(\mathbf{v}) = \mathit{val}(\mathbf{v}') \wedge \mathit{cost}(\mathbf{v}) \leq \mathit{cost}(\mathbf{v}')$"), we can conclude from the above that:

$$\forall \tau_1,\tau_2 \text{ types}, g :: \tau_1 \to \tau_2, t_1 :: \tau_1, t_2 :: \tau_1.\ [\![g\ (f[\tau_1/\alpha]\ t_1\ t_2)]\!]^{¢}_{\emptyset} \sqsubseteq [\![f[\tau_2/\alpha]\ (g\ t_1)\ (g\ t_2)]\!]^{¢}_{\emptyset}$$

In the interest of readability, we will sometimes blur the distinction between syntax and semantics a bit, and additionally keep type substitution (for instantiating polymorphic functions) silent, so that the above conclusion would be written as simply

$$g\ (f\ t_1\ t_2) \sqsubseteq f\ (g\ t_1)\ (g\ t_2) \tag{5}$$

To emphasize again that we crucially exploit polymorphism, recall from the introduction that a corresponding statement does *not* hold for $f :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$. Even if $[\![f]\!]_{\emptyset} = [\![\lambda x :: \mathsf{Nat}.\lambda y :: \mathsf{Nat}.y]\!]_{\emptyset}$ in the cost-free semantics, there can be $g :: \mathsf{Nat} \to \mathsf{Nat}$ and $t_1, t_2 :: \mathsf{Nat}$ such that, of course, $\mathit{val}([\![g\ (f\ t_1\ t_2)]\!]^{¢}_{\emptyset}) = \mathit{val}([\![f\ (g\ t_1)\ (g\ t_2)]\!]^{¢}_{\emptyset})$ is true, but (5) is false.

Let us now move on to other examples, like (4) in the introduction. First, we define $\mathbf{mapPair} = [\![\mathit{mapPair}]\!]^{¢}_{\emptyset}$ for some reasonable rendering of the *mapPair*-function in our calculus. Then, we can give analogues of Proposition 1 and Corollary 2 for pair-lifting, given sets S_1, S_2, S_3, and S_4, $\mathscr{C}(\cdot)$-lifted functions $\mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1})$ and $\mathbf{h} \in \mathscr{C}(\mathscr{C}(S_4)^{S_3})$, and $\mathscr{C}(\cdot)$-lifted values $\mathbf{x_1},\ldots,\mathbf{x_n} \in \mathscr{C}(S_1)$ and $\mathbf{y_1},\ldots,\mathbf{y_m} \in \mathscr{C}(S_3)$, with $n,m \in \mathbb{N}$.

Proposition 2. *Let* $\mathbf{p} \in \mathscr{C}(S_1 \times S_3)$ *and* $\mathbf{q} \in \mathscr{C}(S_2 \times S_4)$. *Then* $(\mathbf{p},\mathbf{q}) \in \mathit{lift}^{¢}_{(,)}(\mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}), \mathscr{C}(R^{\mathbf{h}}_{\mathbf{y_1},\ldots,\mathbf{y_m}}))$ *if and only if there exist* $i \in \{1,\ldots,n\}$, $j \in \{1,\ldots m\}$, *and* $c \in \mathbb{Z}$ *such that* $\mathbf{p} = c \rhd \mathit{appCost}(\mathbf{mapPair}\ ¢\ (\mathbf{g},\mathbf{h})^{¢}, (\mathbf{x_i},\mathbf{y_j})^{¢}) \rhd (\mathbf{x_i},\mathbf{y_j})^{¢}$ *and* $\mathbf{q} = c \rhd (\mathbf{mapPair}\ ¢\ (\mathbf{g},\mathbf{h})^{¢}\ ¢\ (\mathbf{x_i},\mathbf{y_j})^{¢})$.

Corollary 3. *Let* $\mathbf{p} \in \mathscr{C}(S_1 \times S_3)$ *and* $\mathbf{q} \in \mathscr{C}(S_2 \times S_4)$. *If* $(\mathbf{p},\mathbf{q}) \in \mathit{lift}^{¢}_{(,)}(\mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}), \mathscr{C}(R^{\mathbf{h}}_{\mathbf{y_1},\ldots,\mathbf{y_m}}))$, *then there exist* $i \in \{1,\ldots,n\}$ *and* $j \in \{1,\ldots,m\}$ *such that* $\mathbf{mapPair}\ ¢\ (\mathbf{g},\mathbf{h})^{¢}\ ¢\ \mathbf{p} = \mathit{appCost}(\mathbf{mapPair}\ ¢\ (\mathbf{g},\mathbf{h})^{¢}, (\mathbf{x_i},\mathbf{y_j})^{¢}) \rhd \mathbf{q}$.

Now we can deal with example types involving pairs.

Example 3. Let some term f be given with $\alpha \vdash f :: \alpha \to (\alpha,\alpha)$. By Corollary 1 we have:

$$\forall R \in \mathit{Rel}.\ ([\![f]\!]^{¢}_{\emptyset}, [\![f]\!]^{¢}_{\emptyset}) \in \Delta^{¢}_{\alpha \to (\alpha,\alpha),[\alpha \mapsto R]}$$

By the definition of the logical relation and specialization of R, this gives:

$$\begin{array}{l}\forall S_1,S_2 \text{ sets}, \mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1}), \mathbf{x_1} \in \mathscr{C}(S_1).\\ \quad \forall (\mathbf{x},\mathbf{y}) \in \mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1}}).\ ([\![f]\!]^{¢}_{\emptyset}\ ¢\ \mathbf{x}, [\![f]\!]^{¢}_{\emptyset}\ ¢\ \mathbf{y}) \in \mathit{lift}^{¢}_{(,)}(\mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1}}), \mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1}}))\end{array}$$

From this follows, by Proposition 1 and Corollary 3:

$$\begin{array}{l}\forall S_1,S_2 \text{ sets}, \mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1}), \mathbf{x_1} \in \mathscr{C}(S_1).\\ \mathbf{mapPair}\ ¢\ (\mathbf{g},\mathbf{g})^{¢}\ ¢\ ([\![f]\!]^{¢}_{\emptyset}\ ¢\ (\mathit{appCost}(\mathbf{g},\mathbf{x_1}) \rhd \mathbf{x_1}))\\ = \mathit{appCost}(\mathbf{mapPair}\ ¢\ (\mathbf{g},\mathbf{g})^{¢}, (\mathbf{x_1},\mathbf{x_1})^{¢}) \rhd ([\![f]\!]^{¢}_{\emptyset}\ ¢\ (\mathbf{g}\ ¢\ \mathbf{x_1}))\end{array}$$

which due to the certainly nonnegative difference $\mathit{appCost}(\mathbf{mapPair}\ ¢\ (\mathbf{g},\mathbf{g})^{¢}, (\mathbf{x_1},\mathbf{x_1})^{¢}) - \mathit{appCost}(\mathbf{g},\mathbf{x_1})$ simplifies to:

$$\forall \tau_1,\tau_2 \text{ types}, g :: \tau_1 \to \tau_2, t :: \tau_1.\ f\ (g\ t) \sqsubseteq \mathit{mapPair}\ (g,g)\ (f\ t)$$

Example 4. Let some term f be given with $\alpha \vdash f :: (\alpha,\alpha) \to \alpha$. Using Corollary 1, the definition of the logical relation, and Proposition 2 and Corollary 2 for $R^{\mathbf{g}}_{\mathbf{x_1},\mathbf{x_2}}$, we get:

$$\begin{aligned}
&\forall S_1,S_2 \text{ sets}, \mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1}), \mathbf{x_1},\mathbf{x_2} \in \mathscr{C}(S_1).\ \exists c \in \{\mathit{appCost}(\mathbf{g},\mathbf{x_1}), \mathit{appCost}(\mathbf{g},\mathbf{x_2})\}.\\
&\ \mathbf{g} \mathbin{¢} (\llbracket f \rrbracket^{¢}_{\emptyset} \mathbin{¢} \mathit{appCost}(\mathbf{mapPair} \mathbin{¢} (\mathbf{g},\mathbf{g})^{¢}, (\mathbf{x_1},\mathbf{x_2})^{¢}) \rhd (\mathbf{x_1},\mathbf{x_2})^{¢})\\
&\ = c \rhd (\llbracket f \rrbracket^{¢}_{\emptyset} \mathbin{¢} (\mathbf{mapPair} \mathbin{¢} (\mathbf{g},\mathbf{g})^{¢} \mathbin{¢} (\mathbf{x_1},\mathbf{x_2})^{¢}))
\end{aligned}$$

and thus:

$$\forall \tau_1,\tau_2 \text{ types}, g :: \tau_1 \to \tau_2, t :: (\tau_1,\tau_1).\ g\ (f\ t) \sqsubseteq f\ (\mathit{mapPair}\ (g,g)\ t)$$

In order to also be able to deal with example types involving lists, we define $\mathbf{mapList} = \llbracket \mathit{mapList} \rrbracket^{¢}_{\emptyset}$ for *mapList* as given in Section 2. Then, we give analogues of Propositions 1/2 and Corollaries 2/3, given sets S_1 and S_2, a $\mathscr{C}(\cdot)$-lifted function $\mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1})$, and $\mathscr{C}(\cdot)$-lifted values $\mathbf{x_1},\ldots,\mathbf{x_n} \in \mathscr{C}(S_1)$, with $n \in \mathbb{N}$.

Proposition 3. *We have* $(\mathbf{xs},\mathbf{ys}) \in \mathit{lift}^{¢}_{[]}(\mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}))$ *if and only if there exist* $m \in \mathbb{N}$, $i_1,\ldots,i_m \in \{1,\ldots,n\}$, *and* $c \in \mathbb{Z}$ *such that* $\mathbf{xs} = c \rhd \mathit{appCost}(\mathbf{mapList} \mathbin{¢} \mathbf{g}, [\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}]^{¢}) \rhd [\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}]^{¢}$ *and* $\mathbf{ys} = c \rhd (\mathbf{mapList} \mathbin{¢} \mathbf{g} \mathbin{¢} [\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}]^{¢})$.

Corollary 4. *If* $(\mathbf{xs},\mathbf{ys}) \in \mathit{lift}^{¢}_{[]}(\mathscr{C}(R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}))$, *then there exist* $m \in \mathbb{N}$ *and* $i_1,\ldots,i_m \in \{1,\ldots,n\}$ *such that* $\mathbf{mapList} \mathbin{¢} \mathbf{g} \mathbin{¢} \mathbf{xs} = \mathit{appCost}(\mathbf{mapList} \mathbin{¢} \mathbf{g}, [\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}]^{¢}) \rhd \mathbf{ys}$ *and* $\mathit{val}([\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}]^{¢}) = \mathit{val}(\mathbf{xs})$.

Note that the final conclusion in the corollary, $\mathit{val}([\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}]^{¢}) = \mathit{val}(\mathbf{xs})$, keeps a bit more information than we have cared to keep in Corollaries 2 and 3. The reason is that this information will be useful in Example 6 below.

Example 5. Let some term f be given with $\alpha \vdash f :: [\alpha] \to \mathsf{Nat}$. Using Corollary 1, the definition of the logical relation, and Proposition 3 for $R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}$, we get:

$$\begin{aligned}
&\forall S_1,S_2 \text{ sets}, \mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1}), n \in \mathbb{N}, \mathbf{x_1},\ldots,\mathbf{x_n} \in \mathscr{C}(S_1).\\
&\llbracket f \rrbracket^{¢}_{\emptyset} \mathbin{¢} \mathit{appCost}(\mathbf{mapList} \mathbin{¢} \mathbf{g}, [\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢}) \rhd [\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢} = \llbracket f \rrbracket^{¢}_{\emptyset} \mathbin{¢} (\mathbf{mapList} \mathbin{¢} \mathbf{g} \mathbin{¢} [\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢})
\end{aligned}$$

and thus:

$$\forall \tau_1,\tau_2 \text{ types}, g :: \tau_1 \to \tau_2, t :: [\tau_1].\ f\ t \sqsubseteq f\ (\mathit{mapList}\ g\ t)$$

Example 6. Let some term f be given with $\alpha \vdash f :: [\alpha] \to [\alpha]$. Using Corollary 1, the definition of the logical relation, and Proposition 3 and Corollary 4 for $R^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}$, plus simplification, we get:

$$\begin{aligned}
&\forall S_1,S_2 \text{ sets}, \mathbf{g} \in \mathscr{C}(\mathscr{C}(S_2)^{S_1}), n \in \mathbb{N}, \mathbf{x_1},\ldots,\mathbf{x_n} \in \mathscr{C}(S_1).\ \exists m \in \mathbb{N}, i_1,\ldots,i_m \in \{1,\ldots,n\}.\\
&\ \mathit{appCost}(\mathbf{mapList} \mathbin{¢} \mathbf{g}, [\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢}) \rhd (\mathbf{mapList} \mathbin{¢} \mathbf{g} \mathbin{¢} (\llbracket f \rrbracket^{¢}_{\emptyset} \mathbin{¢} [\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢}))\\
&\ = \mathit{appCost}(\mathbf{mapList} \mathbin{¢} \mathbf{g}, [\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}]^{¢}) \rhd (\llbracket f \rrbracket^{¢}_{\emptyset} \mathbin{¢} (\mathbf{mapList} \mathbin{¢} \mathbf{g} \mathbin{¢} [\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢}))\\
&\ \wedge \mathit{val}([\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}]^{¢}) = \mathit{val}(\llbracket f \rrbracket^{¢}_{\emptyset} \mathbin{¢} [\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢})
\end{aligned}$$

In order to continue now and derive a statement about the relative efficiencies of *mapList* $g\ (f\ t)$ and $f\ (\mathit{mapList}\ g\ t)$, for types τ_1,τ_2, function $g :: \tau_1 \to \tau_2$, and list $t :: [\tau_1]$, we would need further information about $\mathit{appCost}(\mathbf{mapList} \mathbin{¢} \mathbf{g}, [\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢})$ and $\mathit{appCost}(\mathbf{mapList} \mathbin{¢} \mathbf{g}, [\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}]^{¢})$. This cannot be provided generally, but a number of useful observations is possible. For example, we know that the elements $\mathbf{x_{i_1}},\ldots,\mathbf{x_{i_m}}$ form a subset of $\{\mathbf{x_1},\ldots,\mathbf{x_n}\}$, and hence that evaluation of *mapList* $g\ (f\ t)$ does not incur g-costs on elements other than those already encountered during evaluation of $f\ (\mathit{mapList}\ g\ t)$, though of course a different selection and multiplicities are possible. Moreover, if we assume that g (actually, $\mathbf{g}$) is equally costly on every element of t (on every $\mathbf{x_i}$), or indeed on every term of type τ_1 (on every element of $\mathscr{C}(S_1)$), then we can reduce the question about the relative efficiency of *mapList* $g\ (f\ t)$ and $f\ (\mathit{mapList}\ g\ t)$ to one about the relative length of t and $f\ t$, to which an answer might be known statically by some separate analysis. Also, note that with some extra effort it would even have been possible to explicitly get our hands at the existentially quantified m and $i_1,\ldots,i_m$, namely to establish that $[\mathbf{i_1},\ldots,\mathbf{i_m}] = \mathit{val}(\llbracket f\rrbracket^{¢}_{\emptyset}\ ¢\ ([\mathbf{1},\ldots,\mathbf{n}],0))$.

Let us also briefly comment on applying our machinery to an automatic program transformation that is used in a production compiler (Gill et al. 1993, though in a call-by-need setting, the mainstream Glasgow Haskell Compiler). The cost-insensitive content of the underlying "short-cut fusion" rule, typically proved via a standard free theorem, can be expressed in our setting as follows, for every choice of types τ and τ', polymorphic function $g :: (\tau \to \alpha \to \alpha) \to \alpha \to \alpha$, and $k :: \tau \to \tau' \to \tau'$ and $z :: \tau'$:

$$\mathit{val}(\llbracket \mathbf{lfold}(k,z,g[[\tau]/\alpha]\ (\lambda x :: \tau.\lambda xs :: [\tau].x : xs)\ []_\tau)\rrbracket^{¢}_{\emptyset}) = \mathit{val}(\llbracket g[\tau'/\alpha]\ k\ z\rrbracket^{¢}_{\emptyset})$$

The desirable statement, and certainly the intuitive assumption by which application of short-cut fusion in a compiler is usually justified, would be:

$$\llbracket \mathbf{lfold}(k,z,g[[\tau]/\alpha]\ (\lambda x :: \tau.\lambda xs :: [\tau].x : xs)\ []_\tau)\rrbracket^{¢}_{\emptyset} \sqsupseteq \llbracket g[\tau'/\alpha]\ k\ z\rrbracket^{¢}_{\emptyset} \tag{6}$$

We could even hope to quantify the $c \geq 0$ such that $\llbracket \mathbf{lfold}(k,z,\ldots)\rrbracket^{¢}_{\emptyset} = c \triangleright \llbracket g[\tau'/\alpha]\ k\ z\rrbracket^{¢}_{\emptyset}$ holds, possibly expressing c in terms of the length of the intermediate list $\mathit{val}(\llbracket g[[\tau]/\alpha]\ (\lambda x :: \tau.\lambda xs :: [\tau].x : xs)\ []_\tau\rrbracket^{¢}_{\emptyset})$. But, maybe surprisingly, (6) does *not* actually hold in general. The reason is that g may "use" its arguments for other things than for creating its output. For example, with $\tau = \mathsf{Nat}$, g could be the function $\lambda k :: \mathsf{Nat} \to \alpha \to \alpha.\lambda z :: \alpha.(\lambda x :: \alpha.z)\ (k\ 5\ z)$. Then:

1. On the one hand, $\mathbf{lfold}(k,z,\ldots)$ incurs no costs at all from applying a concrete $k :: \mathsf{Nat} \to \tau' \to \tau'$ to any values, because $g[[\mathsf{Nat}]/\alpha]$ is only applied to $(\lambda x :: \mathsf{Nat}.\lambda xs :: [\mathsf{Nat}].x : xs)$ and $[]_{\mathsf{Nat}}$ during its evaluation, leading to the empty list as intermediate result which is then processed by the **lfold**.
2. On the other hand, $g[\tau'/\alpha]\ k\ z$ does incur costs for evaluating the application $k\ 5\ z$, even though the resulting value is eventually discarded in $(\lambda x :: \alpha.z)\ (k\ 5\ z)$. Moreover, since we are free to choose k (and z) however we want, we are certainly free to make that application $k\ 5\ z$ arbitrarily more costly than the corresponding application $(\lambda x :: \mathsf{Nat}.\lambda xs :: [\mathsf{Nat}].x : xs)\ 5\ []_{\mathsf{Nat}}$ contributing to the cost of 1. above.

Hence, the right-hand side of (6) can be made arbitrarily more costly than its left-hand side. (The same behavior can be provoked in Haskell using the *seq*-primitive.) It is possible to constrain g in such a way that (6) actually holds, and indeed all "reasonable" functions to be used in short-cut fusion can be expected to satisfy the condition thus imposed on g, but spelling out the details is left for future work.

6 Conclusion

We have developed a notion of relational parametricity that incorporates information about call-by-value evaluation costs, and thus allows to derive quantitative statements about runtime from function types. The mechanics of deriving statements that way are a bit more involved than in the purely extensional setting, but we are optimistic that automation like for `http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi` (Böhme 2007) is possible here as well.

As already mentioned, the exact way in which we assign costs to different program constructs does not appear to impact the overall approach much. Hence, we could also work with more detailed and realistic measures, as for example in the work of Liu and Gómez (2001). Of course, we are also interested in moving from a call-by-value setting to a call-by-name/need one, and in extending the results for our calculus to a calculus with general recursion.

7 References

J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *European Symposium on Programming, Proceedings*, volume 6012 of *LNCS*, pages 125–144. Springer, 2010a. doi: 10.1007/978-3-642-11957-6_8.

J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *International Conference on Functional Programming, Proceedings*, pages 345–356. ACM, 2010b. doi: 10.1145/1932681.1863592.

B. Bjerner and S. Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 157–165. ACM, 1989. doi: 10.1145/99370.99382.

S. Böhme. Free theorems for sublanguages of Haskell. Master's thesis, Technische Universität Dresden, 2007.

J. Christiansen, D. Seidel, and J. Voigtländer. Free theorems for functional logic programs. In *Programming Languages meets Program Verification, Proceedings*, pages 39–48. ACM, 2010. doi: 10.1145/1707790.1707797.

A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM, 1993. doi: 10.1145/165180.165214.

P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM, 2004. doi: 10.1145/982962.964010.

J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer, 1996. doi: 10.1007/3-540-61055-3_38.

Y.A. Liu and G. Gómez. Automatic accurate cost-bound analysis for high-level languages. *IEEE Transactions on Computers*, 50(12):1295–1309, 2001. doi: 10.1109/TC.2001.970569.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

M. Rosendahl. Automatic complexity analysis. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 144–156. ACM, 1989. doi: 10.1145/99370.99381.

D. Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995. doi: 10.1093/logcom/5.4.495.

F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer, 2009. doi: 10.1007/978-3-642-02273-9_22.

J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *International Conference on Functional Programming, Proceedings*, pages 124–132. ACM, 2002. doi: 10.1145/583852.581491.

J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM, 2008. doi: 10.1145/1328897.1328445.

J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM, 2009a. doi: 10.1145/1594834.1480904.

J. Voigtländer. Free theorems involving type constructor classes. In *International Conference on Functional Programming, Proceedings*, pages 173–184. ACM, 2009b. doi: 10.1145/1631687.1596577.

P. Wadler. Strictness analysis aids time analysis. In *Principles of Programming Languages, Proceedings*, pages 119–132. ACM, 1988. doi: 10.1145/73560.73571.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404.

A Proof Sketch of Theorem 2

The proof is by induction over the typing derivation, i.e., we have to consider the derivation rules in Figure 2. In the proof we use the same names for the environments as in Theorem 2 (i.e., ρ, σ_1, σ_2) and assume the conditions on them that are given in Theorem 2 are satisfied. We show just three cases.

In the case

$$\Gamma, x :: \tau \vdash x :: \tau$$

the second condition in Theorem 2 ensures that $(\sigma_1(x), \sigma_2(x)) \in \Delta'_{\tau,\rho}$ and hence it holds that $([\![x]\!]^{¢}_{\sigma_1}, [\![x]\!]^{¢}_{\sigma_2}) = ((\sigma_1(x), 0), (\sigma_2(x), 0))$ is in $\mathscr{C}(\Delta'_{\tau,\rho})$.

In the case

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1.t) :: \tau_1 \to \tau_2}$$

we have

$$\begin{aligned}
&([\![\lambda x :: \tau_1.t]\!]^{¢}_{\sigma_1}, [\![\lambda x :: \tau_1.t]\!]^{¢}_{\sigma_2}) \in \mathscr{C}(\Delta'_{\tau_1 \to \tau_2,\rho})\\
\Leftrightarrow&((\lambda \mathbf{v}.1 \rhd [\![t]\!]^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, 0), (\lambda \mathbf{v}'.1 \rhd [\![t]\!]^{¢}_{\sigma_2[x \mapsto \mathbf{v}']}, 0)) \in \mathscr{C}(\Delta'_{\tau_1 \to \tau_2,\rho})\\
\Leftrightarrow&(\lambda \mathbf{v}.1 \rhd [\![t]\!]^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, \lambda \mathbf{v}.1 \rhd [\![t]\!]^{¢}_{\sigma_2[x \mapsto \mathbf{v}]}) \in \Delta'_{\tau_1 \to \tau_2,\rho}\\
\Leftrightarrow&\forall (\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau_1,\rho}.\ (1 \rhd [\![t]\!]^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, 1 \rhd [\![t]\!]^{¢}_{\sigma_2[x \mapsto \mathbf{v}']}) \in \mathscr{C}(\Delta'_{\tau_2,\rho})\\
\Leftrightarrow&\forall (\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau_1,\rho}.\ ([\![t]\!]^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, [\![t]\!]^{¢}_{\sigma_2[x \mapsto \mathbf{v}']}) \in \mathscr{C}(\Delta'_{\tau_2,\rho})
\end{aligned}$$

where the last line is the induction hypothesis.

In the case

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

we reason as follows:

$$\begin{aligned}
&([\![t_1\ t_2]\!]^{¢}_{\sigma_1}, [\![t_1\ t_2]\!]^{¢}_{\sigma_2}) \in \mathscr{C}(\Delta'_{\tau_2,\rho})\\
\Leftrightarrow&([\![t_1]\!]^{¢}_{\sigma_1}\ ¢\ [\![t_2]\!]^{¢}_{\sigma_1}, [\![t_1]\!]^{¢}_{\sigma_2}\ ¢\ [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathscr{C}(\Delta'_{\tau_2,\rho})\\
\Leftarrow&\forall (\mathbf{x}, \mathbf{y}) \in \mathscr{C}(\Delta'_{\tau_1,\rho}).\ ([\![t_1]\!]^{¢}_{\sigma_1}\ ¢\ \mathbf{x}, [\![t_1]\!]^{¢}_{\sigma_2}\ ¢\ \mathbf{y}) \in \mathscr{C}(\Delta'_{\tau_2,\rho})\\
\Leftarrow&([\![t_1]\!]^{¢}_{\sigma_1}, [\![t_1]\!]^{¢}_{\sigma_2}) \in \mathscr{C}(\Delta'_{\tau_1 \to \tau_2,\rho})
\end{aligned}$$

The last line is the first induction hypothesis, the last implication is by Lemma 2, and the second last implication by the second induction hypothesis.