Size Analysis of Higher-Order Functions

Attila Gobi [⋆1], Olha Shkaravska[2], and Marko van Eekelen[⋆⋆2,3]

[1] Faculty of Informatics, Eötvös Loránd University Budapest
[2] Institute for Computing and Information Sciences, Radboud University Nijmegen
[3] Open University of the Netherlands, Heerlen

Abstract. We present a lambda-calculus that formalizes the relations between the sizes of arguments and the sizes of the corresponding results of functions in a higher-order polymorphic strict functional language. On top of usual constructions we consider two operators for finite maps: List, that defines a (higher-order) finite maps, and Shift. Intuitively, size expressions are abstract interpretations of programs in the natural arithmetic.
To prove normalization and diamond (modulo integer axiomatics) property of the calculus we show that it can be expressed in System F.

1 Introduction

We present a calculus that formalizes the relations between the sizes of arguments and the sizes of the corresponding results of functions in a higher-order polymorphic strict functional language. Informally, the calculus extends the lambda-calculus with arithmetic operations and two operators for finite maps: List, that defines a (higher-order) finite maps, and Shift. To our knowledge, the novelty of our approach is in using finite maps and the two operators above to present size of lists. In the future we will consider possibility to infer polynomial size dependencies for higher-order shapely functions, using polynomial interpolation.

Verification conditions we obtain as the result of syntax-directed stage of type-checking, are (conditional) equations in the combination of three theories: lambda-calculus, integer ring and finite maps. From the theory of finite maps we need the extensionally axiom that looks like

$$f = g \iff Dom(f) = Dom(g) \wedge \forall\, n \in Dom(f).f(n) = g(n)$$

and the rewriting definition of Shift- and List-operators.

This research continues the series of work on size analysis of first-order strict functional languages where annotation inference is based on polynomial interpolation[7]. At the end of this paper we show that similar test-and-interpolate heuristic, which hints possible polynomial dependencies between sizes, is applicable for higher-order functions as well.

⋆ The research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003)
⋆⋆ The second and the third authors are supported by Artemis Joint Undertaking in the CHARTER project, grant-nr. 100039

2 Size calculus

Syntax of expressions in the calculus is given by the grammar on figure 1. We call them *size expressions*.

Now we consider a few simple examples to give an idea behind our formalization. We begin with an integer literal, eg. 42. We assume that it does not have size, so we assign the expression $\mathsf{Unsized}$ to it.

The size of a list is expressed by the combinator List. For instance, the size of the list [1] is given by $\mathsf{List}\,1\,(\lambda x.\,\mathsf{Unsized})$. Here the first argument of List denotes the length of the list while the second is a lambda abstraction expressing the sizes of the elements of the list. λ-bound variable x corresponds to the position of the element in a list. For instance, in the expression $\mathsf{List}\,n\,(\lambda x.e(x))$, $e(n-1)$ represent the size of the head, $e(n-2)$ represents the size of the element next to the head, $e(0)$ represents the length of the tailing element. [4] $e(x)$ is a finite map defined on $0,\ldots,n-1$.

To define *higher-order* size expressions for functions we use two sorts of λ-abstraction. For type list we use Λ abstraction, for other types we use λ.

For example, the size expression for a function of type $\texttt{Int} \to \texttt{Int}$ is $\lambda x.\,\mathsf{Unsized}$, since the size of its result is unsized regardless of its argument. Another simple example is the identity function of type $\alpha \to \alpha$. Its size expression is $\lambda x.x$, which expresses the fact it does not change the size of its argument.

For list arguments we use Λ. In a way, it is the "inverse" of List, as List can be seen as a data constructor of a pair, while Λ is the corresponding pattern matching. In this paper it is defined by the following reduction:

$$(\widehat{\lambda}^x_y.e)(\mathsf{List}\,a\,b) \to (e[x := a, y := b])$$

For an example of applying this definition see the comment after the match-rule in Section 3.

In this way we can present the size expression for the result of a function by giving a size expressions of its arguments. in the following example the function addone takes its argument $l' : L(\texttt{Int})$ and appends 1 to the list:

$$\mathsf{addone}\;l' \;= \mathsf{cons}\,1\,l'$$

The simplest way to express its size dependency is:

$$\widehat{\lambda}^s_f.\,\mathsf{List}(s+1)(\lambda x.\,\mathsf{Unsized})$$

Now the combinator Shift is defined. $\mathsf{Shift}\,e_1\,n\,e_2$ inserts n elements of e_1 before e_2, so the reduction rule for the combinator is:

$$(\mathsf{Shift}\,e_1\,c_1\,e_2)c_2 \to \begin{cases} e_1 c_2 & \text{if } c_2 < c_1 \\ e_2(c_2 - c_1) & \text{otherwise} \end{cases}$$

[4] Note that this enumeration of list elements is "opposes" the traditional in the functional languages enumeration, where the head element has number 0, etc. The enumeration we use is more convenient in our reasoning and, for instance, simplifies significantly the match-rule.

$$\begin{aligned}\text{sizeexpr} ::= {} & c \mid x \mid \text{sizeexpr}\,\mathsf{binop}\,\text{sizeexpr} \\ & \mid \lambda\, x\,.\,\text{sizeexpr} \mid \text{sizeexpr}\,\text{sizeexpr} \\ & \mid \Lambda\, x\, y\,.\,\text{sizeexpr} \mid \mathsf{List} \mid \mathsf{Unsized} \\ & \mid \mathsf{Shift}\end{aligned}$$

Fig. 1. Syntax of Size Expressions

For example, using this combinator we can define the size expression of the usual append function:

$$\widehat{\lambda}^{l}_{f}.\widehat{\lambda}^{m}_{g}.\,\mathsf{List}\,(l+m)\,(\mathsf{Shift}\,g\,m\,f)$$

3 Type system

Our type system does not check or infer types, but relies an underlying type system instead. For the point of view of the size checking we only need a function called Type which can give us the correct type of a function. For this reason and for the sake of readability we omit underlying types in typings at all. Thus τ in a typing $z : \tau$ is a size expression. The equality of two types τ_1 and τ_2 means that they have the same normal form modulo axiomatics of integer rings, if we use the reduction rules of λ-calculus and the reduction definitions of List and Shift. We will also need an equation between expression related to extensionality axiom, which we explain later.

$$\frac{D \vdash \tau = \tau'}{D;\ \Gamma,\ z{:}\tau\ \vdash z{:}\tau'}\ \textsc{Var} \qquad \frac{\begin{array}{c} z \notin dom(\Gamma) \\ D;\ \Gamma\ \vdash e_1{:}\tau_z \\ D;\ \Gamma,\ z{:}\tau_z\ \vdash e_2{:}\tau \end{array}}{D;\ \Gamma\ \vdash \mathsf{let}\ z = e_1\ \mathsf{in}\ e_2{:}\tau}\ \textsc{Let}$$

$$\frac{\begin{array}{c} \mathrm{Type}(f) = \alpha_1 \to \ldots \alpha_m \to \alpha_{m+1} \\ \forall i \in 1 \ldots m : \tau_i = \mathrm{fresh}(\alpha_i) \\ \mathsf{True};\ \Gamma,\ z_1{:}\tau_1,\ \ldots,\ z_m{:}\tau_m,\ \mathsf{f}{:}\tau_\mathsf{f}\ \vdash e_\mathsf{f}{:}\tau_\mathsf{f}\ \tau_1\ \ldots\ \tau_m \\ D;\ \Gamma,\ \mathsf{f}{:}\tau_\mathsf{f}\ \vdash e{:}\tau \end{array}}{D;\ \Gamma\ \vdash \mathsf{letfun}\ \mathsf{f}\ z_1 \ldots z_m\ :\ \tau_\mathsf{f} = e_\mathsf{f}\ \mathsf{in}\ e{:}\tau}\ \textsc{LetFun}$$

The difference between LetFun and Let should be clarified. In the case of Let no formal parameters or recursion allowed, but it does not have explicit size signature – partial inference is used in that rule. LetFun can be recursive and can have formal parameters, but it must be annotated by a size expression and our type system can only check its type.
where

$$\mathrm{fresh}(\alpha) = \begin{cases} \mathsf{List}\,\tau\,(\mathrm{fresh}(\beta)) & \text{if } \alpha = \mathsf{L}(\beta) \\ \mathsf{List}\,\tau\,(\lambda\tau.'\,\mathsf{Unsized}) & \text{if } \alpha = \mathsf{L}(\mathtt{Int}) \\ \mathsf{Unsized} & \text{if } \alpha{=}\mathtt{Int} \\ \tau & \text{otherwise} \end{cases},$$

where τ and τ' are fresh size variables.

We omit the cons and nil rule, because we consider them as predefined functions, so their size expressions are in the context Γ. The size expression of nil is the following.

$$\mathsf{nil} : \mathsf{List}\ 0\ (\lambda x.x)$$

Note that $\lambda x.x$ is arbitrary and should be never evaluated in a correctly typed program. A naive version of the typing for cons is

$$\mathsf{cons}\ : \lambda x.\ \widehat{\lambda}^s_f.\ \mathsf{List}\ (s+1)\ (\mathsf{Shift}\ f\ s\ \{0 \mapsto x\})$$

where $\{0 \mapsto x\}$ is the final map with the domain $\{0\}$ that maps 0 to x. This rule gives an insight, however it is rather semantic and we do not have syntactic tools to define finite maps explicitly. So, we use more general version of the typing:

$$\mathsf{cons}\ : \lambda x.\ \widehat{\lambda}^s_f.\ \mathsf{List}\ (s+1)\ (\mathsf{Shift}\ f\ s\ \lambda y.x),$$

$$\frac{\begin{array}{c} D,\ (\widehat{\lambda}^s_f.l)\tau_l = \mathsf{List}\ 0\ \tau';\ \Gamma,\ l\!:\!\tau\ \vdash e_{\mathsf{nil}}\!:\!\tau \\ hd, tl \notin dom(\Gamma) \qquad \tau_{hd} = (\widehat{\lambda}^s_f.f(s-1))\tau_l \qquad \tau_{tl} = (\widehat{\lambda}^s_f.\mathsf{List}(s-1)f)\tau_l \\ D;\ \Gamma,\ hd\!:\!\tau_{hd},\ l\!:\!\tau_l,\ tl\!:\!\tau_{tl}\ \vdash e_{\mathsf{cons}}\!:\!\tau \end{array}}{\begin{array}{l} D;\ \Gamma,\ l\!:\!\tau_l\ \vdash \mathsf{match}\ l\ \mathsf{with} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \qquad\qquad :\tau \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}\ hd\ tl \Rightarrow e_{\mathsf{cons}} \end{array}}\ \textsc{Match}$$

It is worth to note that for a well-typed function the type τ_l is reduced to the type of the form $\mathsf{List}\ \tau_1\ \tau_2$ and τ_1 is reduced to the integer expression. Then the type of hd, which is $\big(\widehat{\lambda}^s_f.f(s-1)\big)\tau_l$, is reduced to $\tau_2(\tau_1 - 1)$ according to the rewriting definition. Similarly, the type of hd, which is $(\widehat{\lambda}^l_f.\mathsf{List}(l-1)f)\tau_l$ is reduced to $\mathsf{List}(\tau_1 - 1)\tau_2$.

$$\frac{D \vdash \tau\ \tau_1 \ldots \tau_n = \tau'}{D;\ \Gamma,\ f\!:\!\tau,\ x_1\!:\!\tau_1\ \ldots x_n\!:\!\tau_n\ \vdash f x_1 \ldots x_n\!:\!\tau'}\ \textsc{FunApp}$$

3.1 Examples

append

$$\begin{array}{l} \mathsf{append}\ (p,\ q) : \widehat{\lambda}^{s_1}_{f_1}.\widehat{\lambda}^{s_2}_{f_2}.\ \mathsf{List}\ (s_1+s_2)\ (\mathsf{Shift}\ f_2\ s_2\ f_1) = \\ \quad \mathsf{match}\ p\ \mathsf{with} \mid \mathsf{nil} \Rightarrow q \\ \qquad\qquad\qquad\quad \mid \mathsf{cons}\ hd\ tl \Rightarrow \mathsf{let}\ tl' = \mathsf{append}\ tl\ q \\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \mathsf{cons}\ hd\ tl' \end{array}$$

Here p and q is of type $\mathsf{L}(\alpha)$. According to the LetFun rule, we are creating fresh size expressions for the arguments by using the function fresh. Assuming that the fresh size expressions are $\mathsf{List}\ a\ b$ and $\mathsf{List}\ c\ d$ for p and q, respectively, we need to prove the following:

$$\begin{array}{l} \mathsf{True};\ \mathsf{append}\!:\!\widehat{\lambda}^{s_1}_{f_1}.\widehat{\lambda}^{s_2}_{f_2}.\ \mathsf{List}\ (s_1+s_2)\ (\mathsf{Shift}\ f_2\ s_2\ f_1),\ p\!:\!\mathsf{List}\ a\ b,\ q\!:\!\mathsf{List}\ c\ d \\ \quad \vdash \mathsf{match}\ldots\!:\!(\widehat{\lambda}^{s_1}_{f_1}.\widehat{\lambda}^{s_2}_{f_2}.\ \mathsf{List}\ (s_1+s_2)\ (\mathsf{Shift}\ f_2\ s_2\ f_1))\ (\mathsf{List}\ a\ b)\ (\mathsf{List}\ c\ d) \end{array}$$

However it is not necessary, but in the examples we do the reductions of size expressions to make the judgements shorted and more readable:

$$\mathsf{True};\ \mathsf{append} : \widehat{\lambda}^{s_1}_{f_1}.\widehat{\lambda}^{s_2}_{f_2}.\,\mathsf{List}\ (s_1+s_2)\ (\mathsf{Shift}\ f_2\,s_2\,f_1),\ p : \mathsf{List}\ a\,b,\ q : \mathsf{List}\ c\,d \\ \vdash \mathsf{match}\ldots : \mathsf{List}\ (a+c)\ (\mathsf{Shift}\ d\,c\,b)$$

The first step is proving the nil branch:

$$a = 0;\ q : \mathsf{List}\ c\,d\ \vdash q : \mathsf{List}\ (a+c)\ (\mathsf{Shift}\ d\,c\,b)$$

$$a = 0 \vdash c = a + c \wedge i \geq 0 \wedge i < a + c\,d\,i = (\mathsf{Shift}\ d\,c\,b)\,i$$

Analyzing the cases according to the definition of Shift:

$$a = 0 \vdash (i \geq 0)\ \wedge\ (i < a + c)\ \wedge\ (i < c) \Rightarrow d\,i = d\,i$$
$$a = 0 \vdash (i \geq 0)\ \wedge\ (i < a + c)\ \wedge\ (i \geq c) \Rightarrow d\,i = b\,(i - c)$$

$$a = 0 \vdash (i \geq 0)\ \wedge\ (i < a + c)\ \wedge\ (i < c) \Rightarrow \mathsf{True}$$
$$a = 0 \vdash (i \geq 0)\ \wedge\ (i < a + c)\ \wedge\ (i \geq c) \Rightarrow \mathsf{False}$$

To apply the match rule, we need to calculate the size expressions for hd and tl:

$$\tau_{hd} = b\,(a-1) \quad \tau_{tl} = \mathsf{List}\,(a-1)\,b$$

It is a let expression so the next step is to analyze the Let binding (which is two function applications) and infer the size of the variable tl':

$$\frac{\mathsf{True} \vdash \left(\widehat{\lambda}^{s_1}_{f_1}.\widehat{\lambda}^{s_2}_{f_2}.\,\mathsf{List}\ (s_1+s_2)\ (\mathsf{Shift}\ f_2\,s_2\,f_1)\right)(\mathsf{List}\,(a-1)\,b)(\mathsf{List}\ c\,d) = \\ \mathsf{List}\ (a-1+c)\ (\mathsf{Shift}\ d\,c\,b)}{\mathsf{True}; \Gamma\ \vdash \mathsf{append}\,tl\,q : \mathsf{List}\ (a-1+c)\ (\mathsf{Shift}\ d\,c\,b)}\ \mathsf{FunApp}$$

Continuing with the let body:

$$\frac{\mathsf{True} \vdash \left(\lambda x.\widehat{\lambda}^{s}_{f}.\,\mathsf{List}\ (s+1)\ (\mathsf{Shift}\ f\,s\,\lambda y.x)\right) \\ \left(b(a-1)\right)\ \left(\mathsf{List}\ (a-1+c)\ (\mathsf{Shift}\ d\,c\,b)\right) = \\ \mathsf{List}\ (a+c)\ (\mathsf{Shift}\ d\,c\,b)}{\mathsf{True};\ \Gamma,\ tl : \mathsf{List}\ (a-1+c)\ (\mathsf{Shift}\ d\,c\,b) \\ \vdash \mathsf{cons}\,hd'\,tl' : \mathsf{List}\ (a+c)\ (\mathsf{Shift}\ d\,c\,b)}\ \mathsf{FunApp}$$

At the end we have to prove the equation above. After reduction we get:

$$\mathsf{List}\ (a-1+c+1)\ (\mathsf{Shift}\,(\mathsf{Shift}\ d\,c\,b)\,(a-1+c)\,(\lambda y.b(a-1))) = \\ \mathsf{List}\ (a+c)\ (\mathsf{Shift}\ d\,c\,b)$$

It is clear that $a - 1 + c + 1 = a + c$. For the nested part the following cases can be identified:

$$\vdash (i \geq 0)\ \wedge\ (i < a + c)\ \wedge\ (i < a - 1 + c) \Rightarrow (\mathsf{Shift}\ d\,c\,b)i = (\mathsf{Shift}\ d\,c\,b)i$$
$$\vdash (i \geq 0)\ \wedge\ (i < a + c)\ \wedge\ (i \geq a - 1 + c) \Rightarrow b(a-1) = (\mathsf{Shift}\ d\,c\,b)i$$

The first one is a tautology, while the second one can be split into two cases by applying the Shift rule again:

$$\vdash (i \geq 0) \ \wedge \ (i < a + c) \ \wedge \ (i \geq a - 1 + c) \wedge (i < c) \Rightarrow b(a-1) = d\, i$$
$$\vdash (i \geq 0) \ \wedge \ (i < a + c) \ \wedge \ (i \geq a - 1 + c) \wedge (i \geq c) \Rightarrow b(a-1) = b\,(i - c)$$

The first one holds because of the contradiction (eg. $i \geq a - 1 + c$ and $i < c$), while the second one can be reduced to the following (which holds as well):

$$\vdash \forall i \in [0..a + c - 1] : \ (i \geq a - 1 + c) \wedge (i \geq c) \Rightarrow a - 1 = i - c$$

map The map function is higher-order, its first argument is a function and its second argument is a list. It maps the elements of that list with its first argument one-by-one. The interesting part of the type checking is the last step (checking $\mathsf{cons}\, hd'\ tl'$). All others are analogous to the previous example.

$$\begin{array}{l} \mathsf{map}\ (g,\, l) : \lambda x.\widehat{\lambda}^{s}_{f}.\, \mathsf{List}\ \ s\ (\lambda i.x(f\,i)) = \\ \quad \mathsf{match}\ p\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\ \qquad\qquad\qquad \mid \mathsf{cons}\, hd\ tl \Rightarrow \mathsf{let}\ tl' = \mathsf{map}\, g\ tl \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in\ let}\ hd' = g\, hd \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in\ cons}\, hd'\ tl' \end{array}$$

Let's assume that the fresh size variables are a and $\mathsf{List}\ b\, c$, than the type environment before the last step is:

$$\Gamma = \{g : a,\ l : \mathsf{List}\ b\, c,\ hd' : a(c(b-1)),\ tl' : \mathsf{List}\ (b-1)\ (\lambda i.x(c\, i)),\ \ldots\}$$

So we have to show the following:

$$\vdash \big(\lambda x.\widehat{\lambda}^{s}_{f}.\, \mathsf{List}\ (s+1)\ (\mathsf{Shift}\ f\, s\, (\lambda y.x))\big)\ \big(a(c(b-1))\big)\ \big(\mathsf{List}\ (b-1)\ (\lambda i.(a(c\, i)))\big) =$$
$$\mathsf{List}\ \ b\ (\lambda i.a(c\, i))$$
$$\vdash \mathsf{List}\ (b - 1 + 1)\ (\mathsf{Shift}\ (\lambda i.(a(c\, i)))\ (b-1)\ (\lambda y.a(c(b-1))))\big) =$$
$$\mathsf{List}\ \ b\ (\lambda i.a(c\, i))$$

The non-trivial part here is:

$$\vdash (j \geq 0) \ \wedge (j < b) \ \wedge \ j \geq b - 1 \ \Rightarrow (\lambda y.a(c(b-1)))(j - b + 1) = (\lambda i.a(c\, i))j$$
$$\vdash (j \geq 0) \ \wedge (j < b) \ \wedge \ j \geq b - 1 \ \Rightarrow a(c(b-1))) = a(c\, j)$$
$$\vdash (j \geq 0) \ \wedge (j < b) \ \wedge \ j \geq b - 1 \ \Rightarrow b - 1 = j$$

t3 The most interesting question is how can the size expressions handle such a polymorphism when an argument can be a list and even a function. To demonstrate this case we define the following function:

$$\mathsf{t3}\ (g,\, x) : \lambda f.\lambda x.f(f(f\,x)) = g(g(g\,x))$$

It's easy to check the type of this function so it is left for the reader. The interesting part is when we use this function in different kinds of expressions:

$$\mathsf{let}\ t = \mathsf{t3}\ \mathsf{t3}\ \mathsf{in}\ t\ \mathsf{addone}$$

In this example the inferred type for t is

$$f_t = \big(\lambda f.\lambda x.f(f(fx))\big)\big(\lambda f.\lambda x.f(f(fx))\big) \to^* \lambda f.\lambda x.\underbrace{f(f(f\ldots(fx)\ldots)}_{27\text{ applications of }f}$$

Using the fact that addone has type $f_{1:} = \widehat{\lambda}^s_f.\,\mathsf{List}\,(s+1)\,(\lambda x.\,\mathsf{Unsized})$:

$$f_t\ f_{1:} = \lambda x.\underbrace{f_{1:}(f_{1:}\ldots(f_{1:}x)\ldots)}_{27\text{ applications of }f_{1:}}$$

We want to prove that this expression is equal to $\widehat{\lambda}^s_f.\,\mathsf{List}(s+27)(\lambda x.\,\mathsf{Unsized})$. Because of the partial application now we have to decide the equality of two abstractions. Here the equality is proven by a property that reflects extensionality axiom: $\lambda\ .x.f = \lambda\ .x.g$ if and only if $(\lambda\ .x.f)x$ and $(\lambda\ .x.g)x$ are reduced to the same normal form. To continue with our example we apply fresh variables (eg. $\mathsf{List}\ a\,(\lambda y.\,\mathsf{Unsized})$) to both sides of the equation. For the left hand side we get:

$$\begin{aligned}
&(\lambda x.\underbrace{f_{1:}(f_{1:}\ldots(f_{1:}x)\ldots)}_{27\text{ applications of }f_{1:}})(\mathsf{List}\ a\,(\lambda y.\,\mathsf{Unsized})) \to \\
&\to \underbrace{f_{1:}\Big(f_{1:}\ldots\Big((\widehat{\lambda}^s_f.\,\mathsf{List}\,(l+1)\,(\lambda x.\,\mathsf{Unsized}))(\mathsf{List}\ a\,(\lambda y.\,\mathsf{Unsized}))\Big)\ldots\Big)}_{27\text{ applications of }f_{1:}} \to \\
&\to \underbrace{f_{1:}\Big(f_{1:}\ldots\Big(\mathsf{List}\,(a+1)\,(\lambda x.\,\mathsf{Unsized})\Big)\ldots\Big)}_{26\text{ applications of }f_{1:}} \to^* \mathsf{List}\,(a+27)\,(\lambda x.\,\mathsf{Unsized})
\end{aligned}$$

The following two expressions can be checked similarly:

$$\begin{aligned}
&\mathsf{let}\ t = \mathsf{t3}\ \mathsf{addone}\ \mathsf{in}\ \mathsf{t3}\ t && : \widehat{\lambda}^s_f.\,\mathsf{List}\,(s+9)\,\lambda y.\,\mathsf{Unsized} \\
&\mathsf{letfun}\ t\ (x)\ :\ \lambda x.\,\mathsf{Unsized} = x+1\ \mathsf{in}\ \mathsf{t3}\ t && : \lambda x.\,\mathsf{Unsized}
\end{aligned}$$

4 Normalization

Although recursion in size expressions is not allowed, it is easy to express recursive functions using the fixed point combinator ($Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$).

$$\begin{aligned}
&\mathsf{fix}\ (f) : \lambda s.Ys = \\
&\quad \mathsf{let}\ z = \mathsf{fix}\ f \\
&\quad \mathsf{in}\ f\ z
\end{aligned}$$

Even the size expression is entirely correct it is easy to see that we are not able to check it, because the size expression does not have normal form.

Size expressions are defined as a kind of untyped lambda calculus. The easiest way to reason about normalization and diamond property is to give a typed version of our size expressions and give the rewriting rules from the untyped to the typed version of the size calculus. Our choice of type system is System F. However type inference for System F is generally not possible we will show a way how to construct these types from the underlying type system.

The following function transforms an underlying type τ to type of a size expression:

$$SizeType(\tau) = \begin{cases} \forall\alpha.SizeType(a) & \text{if } \tau = \forall\alpha.a \\ SizeType(a) \to SizeType(b) & \text{if } \tau = a \to b \\ L\,a & \text{if } \tau = \mathsf{L}(a) \\ \tau & \text{if } \tau \text{ is a type variable} \\ \mathtt{Unit} & otherwise \end{cases}$$

We assumed the usual `Bool`, `Nat`, `Unit` and product types with the usual operations are defined. The following type is also predefined:

$$L_a := \mathtt{Nat} \times (\mathtt{Nat} \to a)$$

This type expresses the fact that a size of a list is a tuple of the length of the list and a map holding the sizes of the elements of the list. The following table gives some examples:

$\mathtt{Nat} \to \mathtt{Nat}$	$U \to U$
$\mathsf{L}(\mathtt{Nat}) \to \mathsf{L}(\mathtt{Nat})$	$L_U \to L_U$
$\mathsf{L}(a) \to \mathsf{L}(a)$	$\forall a.L_a \to L_a$
$(a \to b) \to \mathsf{L}(a) \to \mathsf{L}(b)$	$\forall a\,b.(a \to b) \to L_a \to L_b$
$\mathsf{L}(a) \to \mathsf{L}(\mathsf{L}(a))$	$\forall a.L_a \to L_{L_a}$

4.1 Types of the size operators

The Unsized can be easy represented by the `Unit` type:

$$\mathsf{Unsized} = \mathsf{unit} : \mathtt{Unit}$$

List corresponds to the data constructor of a pair:

$$\begin{aligned} \mathsf{List} &= \Lambda A.\lambda s^{\mathtt{Nat}}.\lambda f^{\mathtt{Nat}\to A}.\ <s, f> \\ &: \forall A.\mathtt{Nat} \to (\mathtt{Nat} \to A) \to L_A \end{aligned}$$

If $\widehat{\lambda}^s_f.e$ is seen as a syntactic sugar for $\mathsf{Unlist}\ A\,B\,(\lambda sf.e)$, where A and B are types. On the next subsection we will show that it can be inferred. Now it is easy to describe $\widehat{\lambda}^s_f.$ with help of the usual projections π_1 and π_2:

$$\begin{aligned} \mathsf{Unlist} &= \Lambda A.\Lambda B.\lambda f^{\mathtt{Nat}\to(\mathtt{Nat}\to A)\to B}.\lambda t^{L_A}.f\,(\pi^1 t)\,(\pi^2 t) \\ &: \forall A.\forall B.(\mathtt{Nat} \to (\mathtt{Nat} \to A) \to B) \to L_A \to B \end{aligned}$$

The last thing to do is to define the Shift function:

$$\begin{aligned}\mathsf{Shift} &= \Lambda A.\lambda f^{\mathtt{Nat}\to A}.\lambda n^{\mathtt{Nat}}.\lambda g^{\mathtt{Nat}\to A}.\lambda x^{\mathtt{Nat}}.\mathsf{IF}\,A\,(x<n)\,(f\,x)\,(g\,(x-n))\\ &: \forall(A.\mathtt{Nat}\to A)\to\mathtt{Nat}\to(\mathtt{Nat}\to A)\to\mathtt{Nat}\to A\end{aligned}$$

4.2 Type inference

It is easy to see that if the underlying type system is a rank-1 predicative type system then all of our types will be rank-1 predicative. It means that the type reconstruction is possible using some kind of Hindley-Milner type inference. As we can tell the correct type of the size expression for any function we need only partial inference, which means it seams it is also possible to check size expressions if the underlying type system is System F using HMF [5] or MLF [4], however investigation of this possibility is a future plan.

4.3 $(\widehat{\lambda}^{s}_{f}.e_1)e_2$ reduction

We prove that our previously defined reduction rule for $\widehat{\lambda}^{s}_{f}.$ and List is correct eg. every size expression which can be typed is strongly normalizable. This can be done by proving that the original reduction rule and the typed reduction gives the same result. So let e_1 and e_2 be fixed expressions and consider the expression $(\widehat{\lambda}^{s}_{f}.e_1)e_2$ which is assumed to be well-typed. So there are corresponding $\widehat{e_1}$ and $\widehat{e_2}$ sized expressions where e_1 and e_2 can be get by type erasure.

In the untyped system we can assume that e_2 can be reduced to normal form. It must be on the form $\mathsf{List}\,ab$ because of well-typedness, so we apply our rule, and the result of the application will be $e_1[s := a][f := b]$.

The most generic type for $\lambda s.\lambda f.e_1$ is $\mathtt{Nat}\to(\mathtt{Nat}\to B)\to A$, so the desugared expression is: $(\mathsf{Unlist}\;A\,B\,(\lambda s.\lambda f.\widehat{e_1}))\widehat{e_2}$. Because of well-typedness the normal form of $\widehat{e_2}$ is of the form $<\widehat{a},\widehat{b}>$. So the whole expression can be reduced to: $\lambda s.\lambda f.\widehat{e_1}\,\widehat{a}\,\widehat{b}\to_{\beta *}\widehat{e_1}[s := \widehat{a}, f := \widehat{b}]$.

Because of the correspondence between typed and untyped calculus a and b can be get by type erasure. Hence the result of the untyped expression can be got by type erasure. Taking into account that System F is confluent, our embedding of the rule is sane.

4.4 Diamond property of the rewriting system modulo integer ring axiomatics

In this section, instead of $(\mathsf{Shift}\,u_1u_2u_3)u_4$ we consider its desugared definition via branching operator IF.

Based on commutativity and associativity of addition and multiplication, and their distributivity, we introduce the corresponding equivalence relation on terms of our calculus. It is defined by induction on the term structure:

- $c\sim c$, $x\sim x$, $\mathsf{List}\sim\mathsf{List}$, $\mathsf{Unsized}\sim\mathsf{Unsized}$,
- $t_1\sim t_1', t_2\sim t_2' \;\Rightarrow t_1\mathsf{binop}\,t_2\sim t_1'\mathsf{binop}\,t_2'$,

– $t_1, t_2 : \mathtt{Nat} \Rightarrow t_1 + t_2 \sim t_2 + t_1,\ t_1 * t_2 \sim t_2 * t_1$,
– $t_1, t_2, t_3 : \mathtt{Nat} \Rightarrow (t_1 + t_2) + t_3 \sim t_1 + (t_2 + t_3),\ (t_1 * t_2) * t_3 \sim t_1 * (t_2 * t_3)$,
– $t_1, t_2, t_3 : \mathtt{Nat} \Rightarrow t_1 * (t_2 + t_3) \sim t_1 * t_2 + t_1 * t_3$, and the inverse distributivity $t_1 * t_2 + t_1 * t_3 \sim t_1 * (t_2 + t_3)$ holds as well,
– $t \sim t' \Rightarrow \lambda x.\, t \sim \lambda x.t'$,
– $t_1 \sim t_1', t_2 \sim t_2' \Rightarrow t_1 t_2 \sim t_1' t_2'$,
– $t \sim t' \Rightarrow \Lambda x\, y.\, t \sim \Lambda x\, y.t'$,
– $t_1 \sim u_1, t_2 \sim u_2, t_3 \sim u_3 \Rightarrow \mathsf{IF}(t_1, t_2, t_3) \sim \mathsf{IF}(u_1, u_2, u_3)$,
– no other pairs of terms can be added to this relation.

It is an exercise to prove that $t_1 + (t_2 + t_3) \sim t_1 + (t_2 + t_3)$ and $(t_1 + t_2) * t_3 \sim t_1 * t_3 + t_2 * t_3$, and $t_1 * t_3 + t_2 * t_3 \sim (t_1 + t_2) * t_3$. Moreover, by induction on the structure of term t one proves the reflexivity, symmetry and transitivity for $\sim$ (that is the fact that $\sim$ is indeed an equivalence). All these statements are proven in Appendix.

Now, we follow the obvious definition of the diamond property modulo $\sim$ from the paper [6]: the diamond property holds if $\sim \cdot \leftarrow^* \cdot \twoheadrightarrow^* \cdot \sim \subseteq \downarrow_\sim$, where $\cdot$ denotes composition of relations and $\downarrow_\sim$ denotes *joinability modulo* $\sim$ that is the composition $\twoheadrightarrow^* \cdot \sim \cdot \leftarrow^*$. To prove diamond-modulo-$\sim$ property for the calculus, we need first to prove a series technical lemmata. Two substitution lemmata above are proven by induction of the structure of terms in the equivalence relation. See Appendix for the full proofs.

Lemma 1 (Substitutions 1). *If $t_1 \sim t_2$ and x is free in t then $t[x := t_1] \sim t[x := t_2]$.*

Lemma 2 (Substitutions 2). *If $t \sim t'$ and x is free in t and t' then $t[x := t''] \sim t'[x := t'']$.*

In the next lemma we consider interacting of $\sim$ with 1-step reduction.

Lemma 3 (Reduction). *If $t_1 \to t_1'$ and $t_1 \sim t_2$, then there exists t_2' such that $t_2 \to t_2'$ and $t_1' \sim t_2$.*

Proof. By case of reductions.

– We start with β-reduction. Let $t_1 = C_1(\lambda x.\, t_1'')t_1'''C_1'$, where C_1, C_1' are contexts. Then $t_2 = C_2(\lambda x.\, t_2'')t_2'''C_2'$, where $t_1'' \sim t_2''$ and $t_1''' \sim t_2'''$, and the corresponding contexts are equivalent. Moreover, then $t_1' := C_1 t_1''[x := t_1''']C_1'$. We take $t_2' = C_2 t_2''[x := t_2''']C_2' \sim C_2 t_1''[x := t_2''']C_2' \sim C_2 t_1''[x := t_1''']C_2' \sim C_1 t_1''[x := t_1''']C_1' = t_1'$ by the substitution lemmata and the definition of equivalent terms.
– The case for List-pair reduction is similar to β-reduction and proven by the substitution lemmata as well.
– Let $t_1 = C_1(\mathsf{IF}\ \mathit{True}\ u_1\ u_2)C_1'$. Then $t_2 = C_2(\mathsf{IF}\ \mathit{true}\ v_1\ v_2)C_2'$ for some $u_i \sim v_i$ with $i = 1, 2$. In this case $t_1' = u_1 \sim v_1 = t_2'$. The *False*-case is similar.

Next, by induction on the length of reduction chain, one proves

Lemma 4 (Reduction-Closure). *If* $t_1 \to^* t_1'$ *and* $t_1 \sim t_2$*, then there exists* t_2' *such that* $t_2 \to^* t_2'$ *and* $t_1' \sim t_2'$*.*

Now, we prove the diamond-modulo property.

Lemma 5 (Diamond-Modulo-$\sim$ Property). $\sim \cdot \leftarrow^* \cdot \twoheadrightarrow^* \cdot \sim \subseteq \twoheadrightarrow^* \cdot \sim \cdot \leftarrow^*$

Proof. If $(t, t') \in \sim \cdot \leftarrow^* \cdot \twoheadrightarrow^* \cdot \sim$ then there are $(t'', t''') \in \leftarrow^* \cdot \twoheadrightarrow^*$ such that $t \sim t''$ and $t''' \sim t'$. Since the calculus itself has the diamond property, therefore there exists t_1 such that $t'' \twoheadrightarrow^* t_1 \leftarrow^* t'''$. Using the reduction-closure lemma 4, we obtain that there is t_2 such that $t \twoheadrightarrow^* t_2$ and $t_2 \sim t_1$, and there exists t_3 such that $t' \twoheadrightarrow^* t_3$ and $t_3 \sim t_1$. From that follows that $t \twoheadrightarrow^* t_2 \sim t_1 \sim t_3 \leftarrow^* t'$, that is $(t, t') \in \twoheadrightarrow^* \cdot \sim \cdot \leftarrow^*$

5 Related work

Structure of size expressions in our research is close to the approach of A. Abel [1], who has applied sized types for termination analysis of higher-order functional programs. For instance, in his notation sized lists of type A of length $\imath$ are defined as $\lambda\, \imath\, A.\mu^{\imath}.\mathbf{1}+A\times X$ and size expressions are higher-order arithmetic expressions with λ-abstraction as well. The difference is that in that work one uses linear arithmetic over *ordinals*, where ordinals represent zero-order sizes. Moreover, in that research size information is not a stand-alone formalism, but a part of dependent-type system.

In the paper [8] the authors go beyond linear arithmetic. For a given higher-order functional program, they obtain a set of first-order arithmetical constraints over unknown cost functions f. Solving these constraints w.r.t. f gives desired costs of the program. The underlying arithmetic is the arithmetic over naturals, extended with undefined ϵ and unbounded ω values, equipped with a natural linear order. Size expressions admit addition $+$, multiplication $*$ and subtraction of a constant $-n$, thus such expressions are monotonic. Function types are annotated with natural numbers (*latencies*), e.g. $\alpha \xrightarrow{l} \beta$, so it may be conveniently interpreted as an increment in cost consumption, like l clock ticks if the resource of interest is time. Our approach is different in a sense that we aim at expressing size dependencies directly in terms of sizes of inputs, bypassing latencies.

In paper [2] the authors approach to complexity analysis of an imperative language, which is a version of Gödel's T. It is done via abstract interpretation of programs in a semiring of matrices. Informally, matrices represent data flow along program variables. The authors give an upper bound for the return values in term of initial values. However, this is a conjecture and no proof is given. Similarly the conjecture about existing of an abstract interpretation is not proven.

In recent paper [3] the authors develop amortized cost analysis for a higher-order functional language *Shopenhauer*. The analysis is generic, that is it is applicable to different sorts of resources: heap usage, stack size and the number of function calls. Type-derivation procedure generates linear constraints, solving of which gives desirable upper bounds. The analysis succeeds for sure, if bounds are linear. So far, the methodology does not support polymorphic recursion.

6 Conclusions

We presented a size analysis for higher order functions for a higher-order polymorphic strict functional language. The calculus is based upon the lambda-calculus extending it with arithmetic operations and special operators for finite maps representing size of lists.

We have shown that the extended λ-calculus we have presented is strongly normalizable (for size expressions of well-typed functions), if a normal form for integer expressions is defined.

We are investigating the possibility to use polynomial interpolation [7] to infer size expressions for higher-order functions as well.

Acknowledgments. The authors would like to convey thanks to Christoph Herrmann for the fruitful discussion.

References

1. Abel, A.: A Polymorphic Lambda-Calculus with Sized Higher-Order Types. Ph.D. thesis, Ludwig-Maximilians University, Munich (2006)
2. Avery, J., Kristiansen, L., Moyen, J.Y.: Static complexity analysis of higher order programs. In: van Eekelen, M., Shkaravska, O. (eds.) Proceedings of the First international conference on FOundational and Practical Aspects of Resource Analysis (FOPARA). LNCS, vol. 6324, pp. 84–99. Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1886124.1886130
3. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static determination of quantitative resource usage for higher-order programs. SIGPLAN Not. 45, 223–236 (January 2010), http://doi.acm.org/10.1145/1707801.1706327
4. Le Botlan, D., Rémy, D.: MLF: raising ml to the power of system f. In: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming. pp. 27–38. ICFP '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/944705.944709
5. Leijen, D.: HMF: simple type inference for first-class polymorphism. In: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming. pp. 283–294. ICFP '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1411204.1411245
6. Ohlebusch, E.: Church-rosser theorems for abstract reduction modulo an equivalence relation. In: RTA. pp. 17–31 (1998)
7. Shkaravska, O., van Eekelen, M.C.J.D., van Kesteren, R.: Polynomial size analysis of first-order shapely functions. Logical Methods in Computer Science 5(2) (2009)
8. Vasconcelos, P.B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs. In: Trinder, P., Michaelson, G., Peña, R. (eds.) Revised selected papers of the 15th international symposium on Implementation of Functional Languages (IFL'03). LNCS, vol. 3145, pp. 86–101. Springer-Verlag, Edinburgh, UK, September 8-11, 2003 (2004)

Appendix

Lemma 6 (Associativity 2). $t_1 + (t_2 + t_3) \sim t_1 + (t_2 + t_3)$

Proof. $t_1+(t_2+t_3) \sim (t_2+t_3)+t_1 \sim t_2+(t_3+t_1) \sim (t_3+t_1)+t_2 \sim t_3+(t_1+t_2) \sim (t_1 + t_2) + t_3$ due to associativity and commutativity.

Lemma 7 (Distributivity 2). $(t_1+t_2)*t_3 \sim t_1*t_3+t_2*t_3$ *and* $t_1*t_3+t_2*t_3 \sim (t_1 + t_2) * t_3$.

Proof.
- $(t_1+t_2)*t_3 \sim t_3*(t_1+t_2) \sim t_3*t_1+t_3*t_2 \sim t_1*t_3+t_2*t_3$, applying commutativity and distributivity,
- the second equivalence is proved by the symmetric chain.

By induction on the structure of term t we can prove the reflexivity, symmetry and transitivity lemmata (that is the fact that $\sim$ is indeed an equivalence).

Lemma 8 (Reflexivity). $t \sim t$.

Proof.
- If $t = x$ is a variable then $t = x \sim x = t$ follows from the assumption and the equivalence $x \sim x$ by the definition.
- Let $t = t_1 \mathsf{binop}\ t_2$. Then, by induction assumption $t_1 \sim t_1, t_2 \sim t_2$ and by the definition of $\sim$ we have $t = t_1 \mathsf{binop}\ t_2 \sim t_1 \mathsf{binop}\ t_2 = t$.
- If t is given by one of the lambda-abstractions or by the application or by IF, the proof is similar.

Lemma 9 (Symmetry). $t \sim t' \Rightarrow t' \sim t$.

Proof. From the definition of $\sim$ it follows that $t \sim t'$ must be either an instance of integer axiomatics, or (if not) t and t' must be of the same structure (i.e. either both are variables, or both are composed by binop, or by one of two applications, or by abstractions, or by *If*).

- Let the equivalence be an instance of the axioms:
  - if $t = (t_1 + t_2)$ for some t_1, t_2 and $t \sim t'$ is an instance of commutativity then $t' = (t_2 + t_1)$. Therefore, by the definition of $\sim$ (commutativity case) we have $t' = (t_2 + t_1) \sim (t_1 + t_2) = t$,
  - if $t = \big((t_1 + t_2) + t_3)\big)$ for some t_1, t_2, t_3 and $t \sim t'$ is an instance of associativity, then $t' = \big(t_1 + (t_2 + t_3)\big)$. Therefore, by lemma 6 $t' = \big(t_1 + (t_2 + t_3)\big) \sim \big((t_1 + t_2) + t_3)\big) = t$,
  - if $t = \big(t_1 * (t_2 + t_3)\big)$ for some t_1, t_2, t_3 and $t \sim t'$ is an instance of distributivity, then $t' = \big(t_1 * t_2 + t_1 * t_3)\big)$. Therefore, by the inverse distributivity $t' = \big(t_1 * t_2 + t_1 * t_3\big) \sim \big(t_1 * (t_2 + t_3)\big) = t$,
  - if $t \sim t'$ is an instance of the inverse distributivity, the proof of $t' \sim t$ is similar to the proof above.
- Let the equivalence do not follow from the axioms. So, both terms in it are of the same structure.
  - If $t = x$ then t' must be x as well, therefore $t' = x \sim x = t$.

- Let $t = t_1 \mathsf{binop}\, t_2$. Therefore $t' = t'_1 \mathsf{binop}\ t'_2$ as well. Then, by the definition of $\sim$ (and since commutativity is excluded) $t_1 \sim t'_1, t_2 \sim t'_2$. By the induction assumption we have $t'_1 \sim t_1$ and $t'_2 \sim t_2$, therefore $t' = (t'_1 \mathsf{binop}\ t'_2) \ \sim \ (t_1 \mathsf{binop}\ t_2) = t$.
- If t is given by one of the lambda-abstractions or by the application, , or by *If*, the proof is similar.

Lemma 10 (Transitivity). $t \sim t', t' \sim t'' \Rightarrow t \sim t''$.

Proof. From the definition of $\sim$ it follows that $t \sim t'$ must be either an instance of integer axiomatics, or (if not) t and t' must be of the same structure (i.e. either both are variables, or both are composed by binop, or by one of two applications, or by abstractions, or by *If*).

– Let the equivalence be an instance of the axioms:
  - if $t = (t_1 + t_2)$ for some t_1, t_2 and $t \sim t'$ is an instance of commutativity then $t' = (t_2 + t_1)$. Now, we have to do the same analysis for $t' \sim t''$.
    * if $t' \sim t''$ is an instance of commutativity axiom then $t'' = t_1 + t_2 = t$,
    * if $t' \sim t''$ is an instance of associativity axiom then $t' = (t_2 + t_1)$ with $t_2 = (t_{21} + t_{22})$ for some t_{21}, t_{22}. Therefore, $t'' = t_{21} + (t_{22} + t_1) \sim^{lemmaAssociativity2} \sim (t_{21} + t_{22}) + t_1 \sim t_2 + t_1 \sim t_1 + t_2 = t$,
    * if $t' \sim t''$ is an instance of distributivity axiom then it may be only inverse distributivity due to the structure of $t' = (t_2 + t_1)$ and with $t_2 = (t_{11} * t_{22}), t_1 = (t_{11} * t_{12})$ for some t_{11}, t_{12}, t_{22}. Therefore, $t'' = t_{11} * (t_{22} + t_{12}) \sim t_{11} * t_{22} + t_{11} * t_{12} = t_2 + t_1 \sim t_1 + t_2 = t$,
    * if $t' \sim t''$ is an instance of the structure-cases of the definition of $\sim$, then $t'' = t''_2 + t''_1$, where $t''_2 \sim t_2$ and $t''_1 \sim t_1$; therefore $t'' = t''_2 + t''_1 \sim t''_1 + t''_2 \sim t_1 + t_2 = t$.
  - the proof is similar (based induction, axiomatics and derived lemmata, the definition of σ) if the first equivalence is the instance of other axioms.

– Let the first equivalence do not follow from the axioms. So, both terms in it are of the same structure.
  - If $t = x$ then t' must be x as well and the same holds for t'', therefore $t'' = x \sim x = t$.
  - If the second equivalence is an instance of axioms, then the proof is similar to the proof for the pair of equivalences with the axiom being first and the structural case being second, see above.
  - If both equivalences are given by a structural case of the definition of $\sim$, then the proof is straightforward by induction assumption.

Lemma 11 (Substitutions 1). *If $t_1 \sim t_2$ and x is free in t then $t[x := t_1] \sim t[x := t_2]$.*

Proof. – If $t = x$ is a variable then $t[x := t_1] = t_1 \sim t_2 = t[x := t_2]$ follows from the assumption $t_1 \sim t_2$.
– Let $t = (t' \mathsf{binop}\ t'')$. Then $t[x := t_1] = (t'[x := t_1] \mathsf{binop}\ t''[x := t_1]) \sim (t'[x := t_2] \mathsf{binop}\ t''[x := t_2]) = t[x := t_2]$ by the induction assumption.

– If t is given by one of the lambda-abstractions or by the application or by *If*, the proof is similar.

Lemma 12 (Substitutions 2). *If $t \sim t'$ and x is free in t and t' then $t[x := t''] \sim t'[x := t'']$.*

Proof. From the definition of $\sim$ it follows that $t \sim t'$ must be either an instance of integer axiomatics, or (if not) t and t' must be of the same structure (i.e. either both are variables, or both are composed by binop, or by one of two applications, or by abstractions, or by *If*).

– Let the equivalence be an instance of the axioms:
  - if $t = (t_1 + t_2)$ for some t_1, t_2 and $t \sim t'$ is an instance of commutativity then $t' = (t_2 + t_1)$. Therefore, by the definition of $\sim$ (commutativity case) we have $t[x := t''] = (t_1[x := t''] + t_2[x := t''])\ \sim (t_2[x := t''] + t_1[x := t'']) = t'[x := t'']$,
  - if $t = \big((t_1 + t_2) + t_3)\big)$ for some t_1, t_2, t_3 and $t \sim t'$ is an instance of associativity, then $t' = \big(t_1 + (t_2 + t_3)\big)$. Therefore, by the associativity property $t[x := t''] = \big((t_1[x := t''] + t_2[x := t'']) + t_3[x := t'']\big)\ \sim \big(t_1[x := t''] + (t_2[x := t''] + t_3[x := t''])\big) = t'[x := t'']$,
  - if $t = \big(t_1 * (t_2 + t_3)\big)$ for some t_1, t_2, t_3 and $t \sim t'$ is an instance of distributivity, then $t[x := t''] = \big(t_1[x := t''] * (t_2[x := t''] + t_3[x := t''])\big) \sim \big(t_1[x := t''] * t_2[x := t''] + t_1[x := t''] * t_3[x := t'']\big) = \big(t_1 * t_2 + t_1 * t_3\big)[x := t''] = t'[x := t'']$,
  - if $t \sim t'$ is an instance of the inverse distributivity, the proof of $t' \sim t$ is similar to the proof above.

– Let the equivalence do not follow from the axioms. So, they are of the same structure.
  - If $t = x$ then $t' = x$ and $t'[x := t''] = [x := t''] = t'' \sim t'' = [x := t''] = t'[x := t'']$.
  - Let $t = t_1 \mathsf{binop}\ t_2$. Then $t' = t'_1 \mathsf{binop}\ t'_2$ for some t'_1, t'_2. Further, $t[x := t''] = (t_1[x := t'']\mathsf{binop}\ t_2[x := t'']) \sim (t'_1[x := t'']\mathsf{binop}\ t'_2[x := t''] = t'[x := t'']$.
  - If t is given by one of the lambda-abstractions or by the application, , or by *If*, the proof is similar.