# Program Transformations as Proof Transformations for Diadic Logic

Antonina Nepeivoda, Program Systems Institute of RAS, Pereslavl–Zalessky
a_nevod@mail.ru

Given a program on a functional language, one can consider the program as a set of axioms in the intuitionistic logic. The complexity of the function evaluated by the program corresponds to the number of steps in the proof using the axioms. Thus, is the program is transformed to a program with a better asymptotic complexity, the transformation may be considered as a transformation of a longer proof to a shorter one.

For example, let us consider the following program and its logical representation.

$$\begin{array}{ll}
f(0) = 0; & F(0,0) \\
f(S(x)) = S(f(g(S(x)))); & \forall x \forall y \forall z (G(S(x), y) \ \& \ F(y,z) \Rightarrow F(S(x), S(z)))) \\
 & \\
g(0) = 0; & G(0,0) \\
g(S(x)) = h(x); & \forall x, y (H(x,y) \Rightarrow G(S(x), y))) \\
 & \\
h(0) = 0; & H(0,0) \\
h(S(x)) = S(g(x)); & \forall x, y (G(x,y) \Rightarrow H(S(x), S(y)))
\end{array}$$

If we want to prove that $\forall x \exists y, z (H(x,y) \ \& \ F(y,z))$ using the axioms listed above, the obvious way to do that is to split the expression we want to prove into the two parts: $\forall x \exists y (H(x,y))$, and $\forall x \exists y (F(x,y))$. Then these two parts are proved by the induction, and the induction for $F(x,y)$ also may require lemma $\forall x \exists y (G(x,y))$ (or $\forall x \exists y (H(x,y))$).

The proof corresponds to a program that evaluates the call $f(h(x))$ for every natural number $x$. How such a program is built by program transformation tools? A well-known way to do this [1] is to unfold the evaluation tree of the program and then to fold it back into a graph, e. g., as Figure 1 shows. The demonstrated graph is the result of the program transformation tool SPSC [2].

The program transformation tool also splits the evaluation of $f(h(x))$ into an evaluation of $f$ for an arbitrary natural number (which requires an evaluation of $h$), and an evaluation of $g$. Thus, all the three proofs by the induction we assumed in the obvious "human proof" of the corresponding formula of logic are constructed by the program transformation tool automatically. On the evaluation tree, they are shown as the subgraphs containing loops where the induction basis $\sigma(v')$ is looped back to the induction hypothesis $\sigma(v)$.

What techniques allows a program transformation tool to construct "human-like" proof transformations like given above? We try to identify such usual program transformation techniques as using Higman's lemma for termination and the most-specific generalization for splits as mechanisms of the proof transformation in the diadic logic.

**Acknowledgements**

*The research is partially supported by Russian Academy of Sciences, research project No. AAAA-A16-116021760039-0.*

$$\begin{tikzcd}
 & & f(h(x)) \arrow[dll, "x=0"'] \arrow[drr, "x=S(x')"] & & & & \\
f(0) \arrow[d] & & & & \begin{array}{r}\textbf{let } z = S(g(x'))\\ \textbf{in } f(z)\end{array} \arrow[ddllll] \arrow[drr] & & \\
0 & & & & & & f(z) \arrow[dll, "z=0"'] \arrow[d, "z=S(z')"] \\
S(g(x')) \arrow[d, "x=0"'] \arrow[drr, "x'=S(x'')"] & & & & 0 & & \begin{array}{r}\textbf{let } u = g(S(z'))\\ \textbf{in } S(f(u))\end{array} \arrow[dll] \arrow[d] \\
0 & & S(h(x'')) \arrow[dll, "x''=0"'] \arrow[d, "x''=S(x''')"] & & g(S(z')) \arrow[d] & & S(f(u)) \arrow[uu, bend right] \\
0 & & S(S(g(x'''))) \arrow[uull, bend left] & & h(z') \arrow[dl, "z'=0"'] \arrow[d, "z'=S(z'')"] & & \\
 & & & 0 & S(g(z'')) \arrow[dl, "z''=0"'] \arrow[d, "z''=S(z''')"] & & \\
 & & & 0 & S(h(z''')) \arrow[uu, bend right] & &
\end{tikzcd}$$

Figure 1: Evaluation graph of $f(h(x))$

## Bibliography

1. M. H. Sørensen, R. Glück, and N. D. Jones. *A Positive Supercompiler* // Journal of Functional Programming, 1993, vol. 6, pp. 465–479.

2. A Small Positive Supercompiler in Scala *(on-line application)*.
Available at `http://spsc.appspot.com`