To be presented at the 7th International Symposium on Computer and Information Science 1992

Transformational Derivation in the Programming Logic TK

Martin C. Henson,
Department of Computer Science, University of Essex, Colchester, Essex, ENGLAND.
Department of Computer Science, University of Otago, Dunedin, NEW ZEALAND.
hensm@uk.ac.sx

1 Abstract

We examine transformational programming techniques within the programming logic TK. In particular we investigate the transformational technique known as type simulation. The internalisation of this technique within TK is illustrated with respect to a derivation of the α–β-pruning algorithm from a specification of mini-maxing.

2 Introduction

In this paper we investigate transformational programming techniques within the programming logic TK. This theory has been elaborated in detail in [HeT88] and some aspects of program development within TK are explored in [Hen89a]. Our point of departure here is the transformational technique known as type simulation [Hen88] which is a generalisation of a idea due to Wand [Wan80]. We must, for reasons of space, refer the reader to the references for an detailed exposition of the programming logic TK. For the same reason most of the technical results of Section 4 are stated here without detailed proof. The interested reader may like to consult [Hen89c] for these. In Section 3 we describe the rules for inductive types which are central to our formalisation of transformations. The proofs of these propositions can be found in [Hen92]. In Section 4.1 we provide an introduction to transformation via type simulations and their formalisation in TK. Section 4.2 elaborates a more detailed account involving mutual induction and the derivation of the α-β-pruning algorithm.

3 Positive induction

The crucial rules of TK concerning inductive types are as follows:

$$\frac{z \in A}{z \in \Xi(A, \lambda X.B)}\ (\Xi\text{-}intro(i)) \qquad\qquad \frac{z \in B(\Xi(A, \lambda X.B))}{z \in \Xi(A, \lambda X.B)}\ (\Xi\text{-}intro(ii))$$

$$\frac{A \subseteq T \qquad B(T) \subseteq T}{\Xi(A, \lambda X.B) \subseteq T}\ (\Xi\text{-}elim)$$

where X must occur positively in B. Ξ-elim specialises to the following useful induction rule:

$$\frac{(\forall x \in A)\psi(x) \quad (\forall x \in B(\{w \in \Xi(A, \lambda X.B) \mid \psi(w)\}))\psi(x)}{(\forall x \in \Xi(A, \lambda X.B))\psi(x)}$$

The introduction rules Ξ-*intro*(*i*) and Ξ–*intro*(*ii*) are quite intuitive but the Ξ-*elim* rule may require a little motivation. We illustrate it with a simple example: the S-expressions of LISP.

In Miranda™[1] [Tur85] we might write:

sexp *::= Leaf * | Node (sexp *) (sexp *)

In TK this would be translated as follows: $Sexp = \Xi(\{\mathsf{Leaf}\} \times A, \lambda X.\{\mathsf{Node}\} \times X \times X)$ (with A a free type parameter). Ξ-elim specialises in this case to:

$$\frac{(\forall x \in A)\psi(<\mathsf{Leaf}, x>) \quad (\forall x, y \in Sexp)(\psi(x) \wedge \psi(y) \Rightarrow \psi(<\mathsf{Node}, x, y>)}{(\forall x \in Sexp)\psi(x)}$$

which is the expected principle of S-expression induction. In what follows we adopt a Miranda-like notation for the TK types we shall need to define.

3.1 Extracting programs from inductive proofs

Programs are recovered from proofs in TK by means of a modified realizability interpretation [Kle45]. The essential idea is to associate a TK formula $e \; \rho \; \varphi$ with each TK formula φ. $e \; \rho \; \varphi$ is to be read, roughly, as "program e meets specification φ". The full definition is given by induction over the structure of TK formulae. Further motivation, explanation and examples can be found in [Hen89a] and [Hen89b]. Programs are obtained from proofs of such specifications by means of the following, crucial, proposition.

Proposition 3.1.1 *If* TK $\vdash \varphi$ *then there is a* TK *term t for which* TK $\vdash t \; \rho \; \varphi \wedge t\downarrow$

Proof. By induction on the length of the derivation TK $\vdash \varphi$.•

3.2 Mutually inductive systems of induction

Our example in Section 4.2 concerns mutual recursion and this arises from mutually inductive types. We briefly sketch the necessary background. That mutual induction is expressible using a principle of positive induction is well known [Mos74] but what we have to demonstrate is that this can be integrated smoothly with realizability. What we require is that each family of mutually inductive types gives rise to a mutually recursive system of realizing combinators in a natural fashion. For ease of presentation we shall attack binary systems of mutual induction; the generalisation to arbitrary systems is then quite obvious and unenlightening. The outline of the rest of the section is as follows. First we consider a pair of mutually inductive type equations for types T_0 and T_1. We then define a type T so that $T \equiv T_0 \times T_1$. Next we show that there are two elimination rules μ_0 and μ_1 so that $\mu_0 \times \mu_1$ follows from T-elim (similarly for the introduction rules). Finally we define two functions h_0 and h_1 so that $h_0 \times h_1$ is a realizer of T-Elim. Note then that h_0 and h_1 realize the rules μ_0 and μ_1 as a consequence. The reader is referred to [Hen92] for the technical details.

1 Miranda is a trademark of Research Software Ltd.

transformational techniques [Hen88] so it seems to be a good candidate for further investigation in our framework. Higher order accumulator introduction can be specified in TK as follows:

Specification 4.1.1

$(\forall A)(\forall B)(\forall C)(\forall f \in A \to B)(\forall x \in A)(\forall k \in B \to C)(\exists y \in C)(y = k\,(f\,x))$ •

Let us consider one such specialisation of Specification 4.1.1. Take the domain of f to be:
$List(A)$::= Nil | Cons A $List(A)$,
for some type A and $f = (\textbf{\textit{lrec}}\ g\ h)$ for some (suitably typed) g and h.

Specification 4.1.2

$(\forall A)(\forall B)(\forall C)$
$(\forall g \in B)(\forall h \in (List(A) \to B) \to B)(\forall x \in List(A))(\forall k \in B \to C)(\exists y \in C)(y = (k\,(\textbf{\textit{lrec}}\ g\ h\ x)))$

Proof. By $List(A)$-induction. •

Proposition 3.1.1, applied to this proof, provides us with:

$\textbf{\textit{lrek}}\ g\ h\ \text{Nil}\ k \;=\; k\ g$
$\textbf{\textit{lrek}}\ g\ h\ (\text{Cons}\ a\ l)\ k \;=\; \textbf{\textit{lrek}}\ g\ h\ l\ (k \circ (h\ (\text{Cons}\ a\ l)))$
$\textbf{\textit{lrec}}\ g\ h\ l \;=\; \textbf{\textit{lrek}}\ g\ h\ l\ \textbf{\textit{id}}$

As an example of this new combinator we observe that from the following instance of ***lrec***:

$\textbf{\textit{reverse}}\ \text{Nil} \;=\; \text{Nil}$
$\textbf{\textit{reverse}}\ (\text{Cons}\ a\ l) \;=\; (\textbf{\textit{reverse}}\ l) +\!\!+ (\text{Cons}\ a\ \text{Nil})$

we can obtain the program:

$\textbf{\textit{reverse}}\ l \;=\; \textbf{\textit{rev}}\ l\ \textbf{\textit{id}}$
$\textbf{\textit{rev}}\ \text{Nil}\ k \;=\; k\ \text{Nil}$
$\textbf{\textit{rev}}\ (\text{Cons}\ a\ l)\ k \;=\; \textbf{\textit{rev}}\ l\ (k \circ (\lambda z.\ z +\!\!+ (\text{Cons}\ a\ \text{Nil})))$

since $\textbf{\textit{reverse}} = \textbf{\textit{lrec}}\ \text{Nil}\ (\lambda(\text{Cons}\ a\ l).\lambda z.\ z +\!\!+ (\text{Cons}\ a\ \text{Nil}))$.

It is now possible to demonstrate that the program transformation strategy of first order accumulation is subsumed by higher order accumulation and type simulation. The first stage is to find a suitable characterisation of the higher order accumulators appearing in ***rev***. Clearly they are elements of $List(A) \to List(A)$ but we need a finer notion than that. We observe that the accumulators are constructed by the recursive structure of the program in a very systematic fashion. We are led to describe the accumulators by an inductive type:

Definition 4.1.3

$Hacc(X) = \Xi(\{\textbf{\textit{id}}\}, \lambda Y.\{z \mid (\forall y)(\xi(z, y, X) \Rightarrow y \in Y\})$
where $\xi(x, y, X) \Leftrightarrow_{\text{def}} (\exists a \in X)(x = y \circ (\lambda z.z +\!\!+ (\text{Cons}\ a\ \text{Nil})))$ •

Lemma 4.1.4

(*i*) $(\forall X)(Hacc(X) \subseteq List(X) \to List(X))$ (*ii*) $\textbf{\textit{rev}} \in \prod(X, List(X) \to Hacc(X) \to List(X))$

Proof. (*i*) Immediate by $Hacc(X)$-induction. (*ii*) By $List(A)$-induction and Lemma 4.1.4(*i*).

We define a polymorphic relation on $Hacc(X)$ and $List(X)$ as follows:

Definition 4.1.5 $(\forall X)(\forall k \in Hacc(X))(\forall l_0 \in List(X))(k \approx l_0 \Leftrightarrow_{\text{def}} (\forall l_1)((k\ l_1) = l_1 +\!\!+ l_0))$ •

As usual we extend a relation between elements of types to a relation between types by means of:

Definition 4.1.6 Given a binary relation $\mathbf{r} \subseteq A \times B$ we set:

$$A\ \mathbf{r}\ B \Leftrightarrow_{\text{def}} (\forall a \in A)(\exists b \in B)(a\ \mathbf{r}\ b) \wedge (\forall b \in B)(\exists a \in A)(a\ \mathbf{r}\ b)$$ •

Theorem 4.1.7 *(Simulation)*

$(\forall X)(Hacc(X) \approx List(X))$

Proof. We prove the first conjunct by $Hacc(X)$-induction utilising the associativity of $+\!\!+$. The second conjunct is quite elementary: take $x = \lambda z.z +\!\!+\ y$. •

We use the simulation to formulate a specification:

Specification 4.1.8

$(\forall X)(\forall l_0 \in List(X))(\forall k \in Hacc(X))$
$(\exists y \in List(X) \to List(X) \to List(X))(\forall l_1 \in List(X))(k \approx l_1 \Rightarrow (y\ l_0\ l_1) = (\boldsymbol{rev}\ l_0\ k))$

Proof. By $List(X)$-induction and Theorem 4.1.7. •

If we name the program obtained from the proof $\boldsymbol{r}$ we can state:

Corollary 4.1.9 $\boldsymbol{reverse}\ l = \boldsymbol{r}\ l\ \mathsf{Nil}$

Proof. This follows immediately from the base case of Theorem 4.1.7. •

Combining this with the equations available (via Proposition 3.1.1) from the proof of Specification 4.1.8 we finally achieve the expected result:

$$\begin{aligned}
\boldsymbol{reverse}\ l &= \boldsymbol{r}\ l\ \mathsf{Nil}\\
\boldsymbol{r}\ \mathsf{Nil}\ l &= l\\
\boldsymbol{r}\ (\mathsf{Cons}\ a\ l_0)\ l_1 &= \boldsymbol{r}\ l_0\ (\mathsf{Cons}\ a\ l_1)
\end{aligned}$$

4.2 The alpha-beta pruning algorithm

We shall aim to obtain a pair of mutually recursive functions ***prunemax*** and ***prunemin*** each of type $Tree(Nat) \to Nat$ corresponding to the well known α-β-pruning algorithm. Our specification will be based on the concept of mini-maxing. The reader may like to compare our approach with that of [BiH87].

$$\begin{aligned}
T_1 &::= \mathsf{Leaf}\ Nat \mid \mathsf{Node}\ List(T_2)\\
T_2 &::= \mathsf{Leaf}\ Nat \mid \mathsf{Node}\ List(T_1)
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{trec}_1 f_1\, g_1\, (\mathsf{Leaf}\ n) &= f_1\ n\\
\boldsymbol{trec}_1 f_1\, g_1\, (\mathsf{Node}\ z) &= g_1\ (\mathsf{Node}\ z)\ (\boldsymbol{lmap}\ (\boldsymbol{trec}_2 f_2\, g_2)\ z)\\
\boldsymbol{trec}_2 f_2\, g_2\, (\mathsf{Leaf}\ n) &= f_2\ n\\
\boldsymbol{trec}_2 f_2\, g_2\, (\mathsf{Node}\ z) &= g_1\ (\mathsf{Node}\ z)\ (\boldsymbol{lmap}\ (\boldsymbol{trec}_1 f_1\, g_1)\ z)
\end{aligned}$$

We use this to provide our functional specification of mini-maxing.

$$\begin{aligned}
\boldsymbol{evalmax} &= \boldsymbol{trec}_1\ \boldsymbol{id}\ (\lambda z.\lambda v.\ \boldsymbol{reduce}\ \boldsymbol{max}\ {-\infty}\ v)\\
\boldsymbol{evalmin} &= \boldsymbol{trec}_2\ \boldsymbol{id}\ (\lambda z.\lambda v.\ \boldsymbol{reduce}\ \boldsymbol{min}\ \infty\ v)
\end{aligned}$$

We wish these to operate over the type $Tree(Nat)$ where:
$Tree(A) ::= \mathsf{Leaf}\ Nat \mid \mathsf{Node}\ List(Tree(A))$
so we are led to prove the following proposition.

Proposition 4.2.1 *i)* $Tree(Nat) \equiv T_1$ *ii)* $Tree(Nat) \equiv T_2$

Proof. ($\subseteq$) By $Tree(Nat)$-induction and monotonicity of $List(X)$. ($\supseteq$) By (T_1, T_2)-induction. •

Applying higher order accumulation yields two new functions, ***emax*** and ***emin*** as follows:

$$\begin{aligned}
\textbf{\textit{evalmax}}\ t &= \textbf{\textit{emax}}\ t\ \textbf{\textit{id}} \\
\textbf{\textit{evalmin}}\ t &= \textbf{\textit{emin}}\ t\ \textbf{\textit{id}} \\
\textbf{\textit{emax}}\ (\mathsf{Leaf}\ n)\ k &= k\ n \\
\textbf{\textit{emin}}\ (\mathsf{Leaf}\ n)\ k &= k\ n \\
\textbf{\textit{emax}}\ (\mathsf{Node\ Nil})\ k &= k\ {-\infty} \\
\textbf{\textit{emin}}\ (\mathsf{Node\ Nil})\ k &= k\ \infty \\
\textbf{\textit{emax}}\ (\mathsf{Node}\ (\mathsf{Cons}\ t\ l))\ k &= \textbf{\textit{emax}}\ (\mathsf{Node}\ l)\ (k \circ (\textbf{\textit{max}}\ (\textbf{\textit{evalmin}}\ t))) \\
\textbf{\textit{emin}}\ (\mathsf{Node}\ (\mathsf{Cons}\ t\ l))\ k &= \textbf{\textit{emin}}\ (\mathsf{Node}\ l)\ (k \circ (\textbf{\textit{min}}\ (\textbf{\textit{evalmax}}\ t)))
\end{aligned}$$

Following the strategy of Section 4.1 we now look to the accumulators and attempt to find a simulation. These are formed (from ***id***) by functions of the form: $k \circ (\textbf{\textit{min}}\ n)$ or $k \circ (\textbf{\textit{max}}\ n)$ (for some n). Such functions act as restricted identity functions which are bounded above and below. This suggests that accumulators might be simulated by a pair of numbers, representing the lower and upper bounds of corresponding accumulators. To see that this is more than mere conjecture requires the following analysis. In what follows, write ***max*** as an infix $\cup$ and ***min*** as an infix $\cap$.

Definition 4.2.2

$(\forall k \in Nat \to Nat)(\forall n, m \in Nat)$
$(k \approx \langle n, m\rangle \Leftrightarrow_{\text{def}} (n \le m \wedge (\forall s \in Nat)(k\ s = n \cup (s \cap m))))$ •

Fact 4.2.3

(i) $\textbf{\textit{id}} \approx \langle 0, \infty\rangle$ *(ii)* $n \le m \Rightarrow n \cup (s \cap m) = (n \cup s) \cap m$ *(iii)* $n \cup s \cap n = n$ •

Definition 4.2.4

We define an inductive type $\Psi = (\{\textbf{\textit{id}}\}, \lambda X.\{z \mid (\forall y)(\omega(z, y) \Rightarrow y \in X\})$
where: $\omega(x, y) \Leftrightarrow_{\text{def}} (\forall n \in \mathrm{Nat})(x = y \circ (\textbf{\textit{max}}\ n) \vee x = y \circ (\textbf{\textit{min}}\ n))$ •

Lemma 4.2.5 *(i)* $\Psi \subseteq Nat \to Nat$ *(ii)* ***emax***, ***emin*** $\in Tree(Nat) \to \Psi \to Nat$

Proof. *(i)* Immediate by Ψ-induction. *(ii)* By *Tree(Nat)* and Lemma 4.2.5(*i*).

Theorem 4.2.6 *(Simulation)* $\Psi \approx Nat \times Nat$

Proof. Ψ-induction and properties of ***max*** and ***min***. •

Specification 4.2.7

$(\forall t \in Tree(Nat))(\forall k \in \Psi)(\exists y_0, y_1 \in Tree(Nat) \to Nat \to Nat \to Nat)(\forall n, m \in Nat)$
$(k \approx \langle n, m\rangle \Rightarrow (\textbf{\textit{emax}}\ t\ k) = (y_0\ t\ n\ m) \wedge (\textbf{\textit{emin}}\ t\ k) = (y_1\ t\ n\ m))$

Proof. By *Tree(Nat)*-induction and Theorem 4.2.6 •

Let ***prunemax*** and ***prunemin*** be the functions obtained via Proposition 3.1.1. We then have:

Lemma 4.2.8

(i) $\textbf{\textit{prunemax}}\ t\ n\ m = n \cup (\textbf{\textit{evalmax}}\ t) \cap m$ *(ii)* $\textbf{\textit{prunemin}}\ t\ n\ m = n \cup (\textbf{\textit{evalmin}}\ t) \cap m$

Proof. Fact 4.2.3(*ii*) and properties of ***max*** and ***min***. •

Using Lemma 4.2.8 and Fact 4.2.3 we obtain the final program:

$$\begin{array}{lcl}
\textbf{\textit{prunemax}}\ (\mathsf{Leaf}\ a)\ n\ m & = & n \cup a \cap m \\
\textbf{\textit{prunemin}}\ (\mathsf{Leaf}\ a)\ n\ m & = & n \cap a \cup m \\
\textbf{\textit{prunemax}}\ (\mathsf{Node\ Nil})\ n\ m & = & n \\
\textbf{\textit{prunemin}}\ (\mathsf{Node\ Nil})\ n\ m & = & m \\
\textbf{\textit{prunemax}}\ (\mathsf{Node}\ (\mathsf{Cons}\ t\ l))\ n_0\ m & = & \\
\quad \textbf{let}\ n_1 = (\textbf{\textit{prunemin}}\ t\ n_0\ m)\ \textbf{in}\ \textbf{\textit{if}}\ (n_1 = m)\ m\ (\textbf{\textit{prunemax}}\ l\ n_1\ m) & & \\
\textbf{\textit{prunemin}}\ (\mathsf{Node}\ (\mathsf{Cons}\ t\ l))\ n\ m_0 & = & \\
\quad \textbf{let}\ m_1 = (\textbf{\textit{prunemax}}\ t\ n\ m_0)\ \textbf{in}\ \textbf{\textit{if}}\ (\textbf{\textit{eq}}\ m_1\ n)\ n\ (\textbf{\textit{prunemin}}\ l\ n\ m_1) & &
\end{array}$$

5 Acknowledgements

Thanks as usual are owed to the members of the Essex Constructive Set Theory and Functional Programming Group. Part of this work was carried out while I was a visiting fellow in the Department of Computer Science at the University of Otago in New Zealand. I should like to take this opportunity to extend my gratitude to my hosts for making my visit so worthwhile. This work was supported by an SERC research grant: GR/F/02809.

6 References

[BiH87] Bird, R. S. & Hughes, J., **The alpha-beta algorithm: an exercise in program transformation**, *Inf. Proc. Lett.*, **24**, pp 53-57, 1987.

[HeT88] Henson, M. C. & Turner, R., **A constructive set theory for program development**, *Proc. 8th Conf. on FST & TCS*, Bangalore, LNCS **338**, pp 329-347, Springer, 1988.

[Hen88] Henson, M. C., **Higher order transformations and type simulations**, *Comp. J.*, **31**(6), pp 517-524, 1988.

[Hen89a] Henson, M. C., **Program development in the constructive set theory TK**, *Formal Aspects of Computing*, **1**(2), pp 173-192, 1989.

[Hen89b] Henson, M. C., **Realizability models for program construction**, *Proc. Conf. on Mathematics of Program Construction*, Gröningen, LNCS **375**, pp 256-272, Springer, 1989.

[Hen89c] Henson, M. C., **Transformational programming, type simulations and intensional set theory**, *University of Essex, Department of Computer Science report, CSM-121*, 1989.

[Hen92] Henson, M. C., **Safe positive induction in the programming logic TK**, in: *Logic Programming* (ed. Voronkov, A.), LNCS **592**, pp 215-231, Springer, 1992.

[Kle45] Kleene, S. C., **On the interpretation of intuitionistic number theory**, *J. Symb. Logic*, **10**, pp 109-124, 1945.

[Mos74] Moschovakis, Y. N., **Elementary induction on abstract structures**. North Holland, 1974.

[Tur85] Turner, D. A, **Miranda - A non-strict functional language with polymorphic types**, *in: Proc. IFIP Int. Conf. on functional programming languages and computer architecture*, Nancy, LNCS **201**, Springer Verlag, pp 445-472, 1985.

[Wan80] Wand, M., **Continuation based transformation strategies**, *J. ACM*, **27**(1), pp 164-180, 1980.