Program Derivation in PowerEpsilon

Ming-Yuan Zhu and Cheng-Wei Wang

Beijing Institute of Systems Engineering
P.O. Box 9702-19, Beijing 100101, People’s Republic of China

Abstract: In this paper, we present a proof development system called PowerEpsilon, based on a constructive type theory which can be used as a formal program development system for actually deriving a program from a specification.

1. Introduction

The task of computer programming is to write an efficient and correct program for a given specification. Since correctness is one of the major concerns in computer programming, we will need to develop an understanding of the causal connections between software requirement specifications and the computer programs that realize them, much like the understanding that mathematical logic brings to mathematical reasoning. This kind of understanding will help us to formalize the programming process and to build program development system to support this process.

Martin-Löf’s Type Theory [6-7] and the Calculus of Constructions (CC) invented by Coquand and Huet [3] are the formal theory based on constructive mathematics. Both can be used for writing specifications and constructing programs, where a specification is expressed by a type and an element of that type is a program that satisfies the specification. Therefore, in type theory and CC, the causal connections between specifications and programs are clear; the specifications play the role of the propositions to be proved, and the programs are obtained from the proofs.

In this paper, we present a proof development system called PowerEpsilon, based on a constructive type theory which can be used as a formal program development system for actually deriving a program from a specification. A programming exercise is given to show how the system works.

PowerEpsilon, currently developed in our institute, is a polymorphic language based on Martin-Löf’s Type Theory and the Calculus of Constructions. In PowerEpsilon, the concept of limit of type universe hierarchies (kind) and a scheme for inductive define types are introduced. The system can be used as both a programming language with a very rich set of data structures and a metalanguage for formalizing constructive mathematics. The system has been implemented using the software development system AUTOSTAR constructed by first author [8].

PowerEpsilon is a proof checker much similar to other mechanical proof checkers, such as LCF [5] and Nuprl [2], which are completely formal user-controlled systems. However, PowerEpsilon is more powerful than LCF and Nuprl, in which the equality and induction rules for arbitrary inductive types are definable.

2. PowerEpsilon

2.1. Abstract Syntax

The basic expressions of PowerEpsilon, called terms, are defined as follows:

\[
T ::= \text{Prop} \mid \text{Type}(i) \mid (i \in \text{Ends}) \mid \text{Kind} \mid x \\
| !(x : T_1) T_2 | (T_1 \rightarrow T_2) | \forall(x : T_1) T_2 \\
| ?(x : T_1) T_2 | @T_1 T_2 | \langle T_1, T_2 \rangle.
\]

where the symbols !, ? and \ stand for universal quantifiers, existential quantifiers, and lambda abstraction quantifiers. The kinds Prop, Type(i), for all i ∈ Ends and Kind are called type universes. Every kind is assigned a number as its level: \ LEV(Prop) = -1; LEV(Type(i))
2.2. Inference Rules of PowerEpsilon

2.2.1. Type Inference Rules

We now describe the judgement forms and the inference rules of PowerEpsilon.

Contexts. A context \( \Gamma \) is a finite sequence of expressions of form \( x : A \), where \( x \) is a variable and \( A \) is a term. The empty context is denoted by \( <> \). The set of free variables in a context \( \Gamma = x_1 : A_1, \ldots, x_n : A_n \) is defined as \( \text{FV}(\Gamma) = \bigcup_{i=1}^{n} \{x_i\} \cup \text{FV}(A_i) \).

Judgements. A judgement is a form \( \Gamma \vdash M : A \), where \( \Gamma\) is a context, and \( M \) and \( A \) are terms. The intuitive meaning of the judgement is that \( M \) has type \( A \) in context \( \Gamma \). We write \( \Gamma \vdash M : A \) for \( (<> \vdash M : A) \).

Inference Rules. In the following rules, \( \Gamma \) is assumed to be a valid context, where \( K \) and \( IC \) stand for arbitrary kinds, \( i, j \) and \( k \) for natural numbers. The rules for valid contexts are:

\[(\text{Ax})\]
\[\vdash \text{Prop} : \text{Type}(0)\]

\[\text{Var} \]
\[\Gamma, x : A, \Gamma' \vdash \text{Prop} : \text{Type}(0)\]

\[\vdash x : A\]

\[(\text{C})\]
\[\Gamma \vdash \text{Prop} : \text{Type}(0)\]

\[\Gamma, x : A \vdash \text{Prop} : \text{Type}(i + 1)\]

\[\text{(K1)}\]
\[\vdash \text{Prop} : \text{Type}(0)\]

\[\Gamma, x : A \vdash \text{Prop} : \text{Kind}\]

\[\text{(K2)}\]
\[\vdash \text{Prop} : \text{Type}(0)\]

\[\Gamma, x : A \vdash \text{Prop} : \text{Type}(i) : \text{Kind}\]

\[\text{(e1)}\]
\[\vdash \text{P} : \text{Prop}\]

\[\Gamma \vdash \text{I}(x : A) : \text{Prop}\]

\[\Gamma \vdash A : K; \Gamma, x : A \vdash B : \text{Type}(j)\]

\[\begin{array}{l}
\Gamma \vdash (x : A) B : \text{Type}(k) \\
\Gamma \vdash \text{I}(x : A) \ B : \text{Type}(k)
\end{array}\]

where \( K = \text{Prop} \) or \( \text{Type}(i)(i \leq 0) \), and \( k = \text{MAX}(\text{LEV}(K), j) \).

\[\text{(e3)}\]
\[\vdash \text{I}(x : A) B : \text{Kind}\]

where \( K = \text{Kind} \) or \( K' = \text{Kind} \).

\[\text{(->1)}\]
\[\vdash [A \rightarrow P] : \text{Prop}\]

\[\Gamma \vdash [A \rightarrow P] : \text{Prop}\]

\[\text{(->2)}\]
\[\vdash [A \rightarrow B] : \text{Type}(k)\]

where \( K = \text{Prop} \) or \( \text{Type}(i)(i \leq 0) \), and \( k = \text{MAX}(\text{LEV}(K), j) \).

\[\text{(->3)}\]
\[\vdash A : K; \Gamma, x : A \vdash B : K'\]

\[\Gamma \vdash A : K; \Gamma \vdash B : K'\]

\[\text{(Pair)}\]
\[I', M : A'; \Gamma, N : B' ; I', x : A \vdash B: K\]

\[\text{where } A' \ll A, B' \ll B[N/x].\]
Derivations. A derivation of a judgement \( J \) is a finite sequence of judgements \( J_1, ..., J_n \) with \( J_n = J \) such that, for all \( 1 \leq i \leq n \), \( J_i \) is a consequence of some instance of an inference rule whose premises are in \( \{ J_k | k < i \} \). A judgement \( J \) is derivable if there is a derivation of \( J \).

\( \Gamma \)-Terms, \( \Gamma \)-Types and \( \Gamma \)-Propositions. A term \( M \) is called a \( \Gamma \)-term (or well-typed term under \( \Gamma \)) if \( \Gamma \vdash M : T \) for some \( T \). A term \( T \) is called a \( \Gamma \)-type if \( \Gamma \vdash T : K \) for some kind \( K \). A \( \Gamma \)-type \( T \) is called a \( \Gamma \)-proposition if \( \Gamma \vdash T : \text{Prop} \) and called a proper \( \Gamma \)-type otherwise.

2.2.2. Type Conversion Rules

We now introduce another kind of judgement \( \Gamma \vdash M = N \), whose intuitive meaning is that the terms \( M \) and \( N \) denote the same object. Here \( = \) is the smallest congruence over propositions and contexts containing \( \beta \)-conversion. The following is the definition of type conversion rules:

\[
\begin{align*}
\Gamma \vdash M : P; \Gamma \vdash P : \text{Prop}; \Gamma \vdash P = Q & \quad \text{(Type Equal 1)} \\
\Gamma \vdash M : Q & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : P; \Gamma \vdash P : \text{Type}(i); \Gamma \vdash P = Q & \quad \text{(Type Equal 2)} \\
\Gamma \vdash M : Q & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : P; \Gamma \vdash P : \text{Kind}; \Gamma \vdash P = Q & \quad \text{(Type Equal 3)} \\
\Gamma \vdash M : Q & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : N & \quad \text{(Reflexivity)} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M = N & \quad \text{(Symmetry)} \\
\Gamma \vdash M = N; \Gamma \vdash N = P & \quad \text{(Transitivity)} \\
\Gamma \vdash M = P & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash P_1 = P_2; \Gamma, x : P_1 \vdash M_1 = M_2 & \quad \text{(Abst-Equal)} \\
\Gamma \vdash \lambda(x : P_1) M_1 = \lambda(x : P_2) M_2 & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \pi_1 = P_2; \Gamma, x : P_1 \vdash M_1 = M_2 & \quad \text{(->-Equal)} \\
\Gamma \vdash \pi_1 \pi_2 = \pi_2 & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : A & \vdash M : P; \Gamma \vdash M = N; \Gamma \vdash N = \text{Prop} & \quad \text{(App-Equal)} \\
\Gamma \vdash \rho(M, N) : P; \Gamma \vdash M = M_2; \Gamma \vdash N = M_1 & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, \pi : \text{Prop} & \vdash \pi \pi = \pi_2 & \quad \text{(Pair-Equal)} \\
\Gamma \vdash \pi_1 \pi_2 = \pi_1 \pi_2 & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : A & \vdash M : P; \Gamma, x : A \vdash N : A & \quad \text{(\( \beta \)-Conversion)} \\
\Gamma \vdash \rho(\lambda(x : A) M, N) = \rho(M, N/x) & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, x, A : B, M^0 : B, A = B, P : [A \to \text{Prop}] & \vdash N : \rho(P, M^0); & \\
\Gamma, x, A, B, C, P : [A \to \text{Prop}] & \vdash N : \rho(P, M^0) & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \iota(u : A) \ast x := u & \quad \text{(MV Equal)} \\
\end{align*}
\]

where \( R \) is a well ordered relation, \( A \) is an inductively defined type, \( M^0 \) of type \( A \) is a \( R \)-Minimal and \( M' \) is the successor of \( M \) under \( R \) [9].

2.2.3. Cumulativity Relation

The type inclusions between the universes induce the type cumulativity that is syntactically characterized by the cumulativity relation \( < \). The binary relations \( <, i \in \mathbb{N} \) over terms are inductively defined as follows:
1. A << 0 B if and only if one the following holds:
   (a) \( A = B \);
   (b) \( A = \text{Prop} \) and \( B = \text{Type}(i) \) for some \( j \in \mathbb{Q} \); or \( A = \text{Type}(i) \) and \( B = \text{Prop} \) for some \( i < j \);
   (c) \( A = \text{Prop} \) and \( B = \text{Kind} \); or \( A = \text{Type}(i) \) and \( B = \text{Kind} \);
   (d) \( A = \text{Prop} \) and \( B = \text{Kind} \); or \( A = \text{Type}(0) \) and \( B = \text{Kind} \).

2. A << \( i + 1 \) B if and only if one of the following holds:
   (a) \( A << i B \);
   (b) \( A = 1(x : A1) A2 \) and \( B = 1(x : B1) B2 \) for some \( A1 << B1 \) and \( A2 << B2 \);
   (c) \( A = 1(x : A1) A2 \) and \( B = 1(x : B1) B2 \) for some \( A1 << B1 \) and \( A2 << B2 \);
   (e) \( A = \text{Not}(x : A1) A2 \) and \( B = \text{Not}(x : B1) B2 \) for some \( A1 << B1 \) and \( A2 << B2 \);
   (f) \( A = \text{SS}(A1, A2) \) and \( B = \text{SS}(B1, B2) \) for some \( A1 << B1 \) and \( A2 << B2 \);
   (g) \( A = <A1, A2> \) and \( B = <B1, B2> \) for some \( A1 << B1 \) and \( A2 << B2 \).

With the definition given above, the cumulativity relation << is defined as follows:

\[ << = U << i \]

3. The Program Derivation Method

As we have mentioned, programming in PowerEpsilon is like theorem proving in mathematics, where the specifications play the role of the propositions to be proved and the programs are then obtained from the proofs.

We are first given a specification that describes a relation between the input and output of the desired program. The specification does not necessarily suggest any method for computing the output. To construct a program that meets the specification, we prove the existence, for any input object, of an output object that satisfies the specified conditions. The proof is conducted in a background theory that expresses the known properties of the subject domain, and thus describes the primitives of the programming language.

**Quotient-Remainder Problem.** The problem is to find two functions QUOT and REMD such that @(QUOT, x1, x2) = z1 and @(REMD, x1, x2) = z2. Here z1 is the quotient and z2 is the remainder of dividing x1 by x2. The specification specifies the behavior of desired program only for case in which the divider x2 is not zero. The specification of quotient-remainder problem is given as follows:

\[
\text{def Nat_eq} = \{ (x1 : Nat, x2 : Nat) \} \\
\text{def Spec} : ! (x1 : Nat, x2 : Nat) \text{Nat_eq}(x1, x2).
\]

A program that satisfies the specification will give a pair of natural numbers as output. Deriving a program that satisfies the specification is nothing but finding a program which is a member of the type Spec, or, if we think of the specification as a goal, to find a program that achieves the goal.

The intuitive idea behind the proof is the following: we will divide the problem into subproblems according to the input natural numbers x1 and x2 with the cases of x1 < x2 and x1 > = x2. The division will be done inductively using a special kind of induction rule:

(a) The base case, i.e. that @(Less, x1, x2), is solved by simply putting the result equal to @(PROD, x1, x1) which means that z1 = 00 and z2 = x1.

(b) For the induction step, assume that @(Greq, x1, x2) and that we have a proof p showing that @(Nat_eq, @(MINUS, x1, x2), x2), then we can put the result equal to @(PROD, @(SS, @(PJ1, p))), @(PJ2, p)).

A special kind of induction rule for the quotient-remainder problem has to be defined on Nat which has the following form:

\[
\text{dec Rec} : \{ (P : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Type}(0)) \} \\
\text{dec Spec} : \{ (x1 : \text{Nat}, x2 : \text{Nat}) \} \text{Nat_eq}(x1, x2).
\]
Here Rec is in some sense very similar to the "computation invariant" -- an important concept in a program verification approach. To define an induction rule is as much difficult as finding a "loop invariant" in program verification. There is no general method of finding appropriate induction rules in order to complete a proof.

We now start the derivation by induction over x1 and x2. First, we assume that @((Less, x1, x2)), and have to show that @((PROD, 00, x1)), where a E @((Nat_eq, x1, x1))

We have LEM1 and THM1 to achieve this goal:

dec LEM1 : @((Less, x1, x2)) -> @((Equal, Nat, x1, x2))
@(( ADD, @((TIMES, x2, @((P1, @((PROD, 00, x1)))))), @((P2, @((PROD, 00, x1))))))

def THM1 = @((Less, x1, x2)) -> @((Equal, Nat, x1, x2))
@(( ADD, @((TIMES, x2, @((P1, @((PROD, @((SS, x1), x2)))))), @((P2, @((PROD, @((SS, x1), x2))))))

To combine THM1 and THM2, we then obtain theorems THM and extract the final program PROG from THM3, where THM3 : Spec:

def THM3 = @((Less, x1, x2))
@((Rec, @((Equal, Nat, x2, 00))))
@(( PROD, @((SS, @((FST, PI)), @((P1, @((PROD, 00, x1))))))

To combine THM1 and THM2, we then obtain theorems THM and extract the final program PROG from THM3, where THM3 : Spec:

def THM3 = @((Less, x1, x2))
@(( Rec, @((Equal, Nat, x2, 00))))
@(( PROD, @((SS, @((FST, PI)), @((P1, @((PROD, 00, x1))))))

The final program PROG can be divided into two programs QUOT and REMD:

def QUOT = @((Less, x1, x2))
@(( Rec, @((Equal, Nat, x2, 00))))
@(( PROD, @((SS, @((FST, PI)), @((P1, @((PROD, 00, x1))))))

where the function ERR can be viewed as an exception handler.

The proofs and program derivations here have been presented in a machine-readable style. The whole proofs and program derivations have been checked out by the PowerEpsilon system.

4. Related Work

Program derivation is one of several methods
that aid software development that would be improved by more formal techniques. Here we shall mention some of the other software development methods.

- Program Verification, or proving that a given program meets a given specification. The method requires that both the specification and the program be given. This is the oldest of the formal approaches.

- Program Transformation, or making a given program into a more efficient, perhaps less understandable equivalent one.

- Rapid Prototyping, or running a specification directly as a program ensures a potential user that the specification actually does agree with expectations.

- Logic Programming, or executing a program expressed in logic.

- Modification, or altering a given program to reflect changes in its specification or environment.

Many researchers [4] in formal methods for software development do regard programming as primarily an inferential process, but are not at all concerned with automating the task. Rather, they intend to provide intellectual tools for the programmer. On the other hand, many tools for automatic theorem proving exist, such as the system developed by Boyer-Moore [1].

All of these methods rely on inferential techniques, and several of them are less ambitious than full program synthesis. By developing more powerful theorem-proving techniques that are specialized to software-engineering applications, we can make progress in several of these areas at once.

5. Conclusions

We have presented here a formal derivation of program for the quotient-remainder problem. The proof method we have used is the backward or goal-directed method. In this method, we start with the specification we want to prove (the goal) and then successively reduce it to simpler subgoals, and a validation which is a justification that we can achieve the original goal if and only if we can achieve the subgoals.

PowerEpsilon can be used as a powerful semantics-based programming tool for development of provably correct software. It has created an entirely new attitude towards the processes of program development, validation and adaptation. It provide natural causal connections between software specifications and computer programs that realize them where specifications are treated as propositions (types) and programs are treated as proofs.

Further investigations and experiments are needed to use PowerEpsilon for developing well-defined, domain-specific problems, e.g., hardware circuits, microprocessors, operating system kernels and compiler systems.

References


