

Introduction to Expression Refinement

Alexander Bunkenburg and Joseph Morris*
Department of Computing Science, University of Glasgow,
jmm∩*bunkenba*@dcs.gla.ac.uk

January 28, 1998

Abstract

We present an outline of a refinement calculus for strict functional programming, based on four-valued logic and equational reasoning. The language imitates the features known from the imperative Refinement Calculus in a language of expressions.

1 Introduction

A refinement calculus provides a formal language and a set of transformations of the language terms. We transform terms which self-evidently describe the desired result (*specifications*) into terms which admit of mechanical interpretation (*programs*). The transformation is known as *refinement*.

Each refinement step is carried out with the degree of formality we deem appropriate: the choice is governed by criteria such as the complexity of the problem domain, the extent to which we can rely on testing, and the tolerances acceptable to the application.

The Refinement Calculus (with capitals) is such a system. It has been developed in [Bac80, Mor87, Mor88, Mor94] and is used to specify and derive imperative programs. However, it has theoretical and practical weaknesses. Firstly, it does not come equipped with a logic appropriate to its use; for example, it does not say how we should reason about partial expressions. Secondly, it imposes a formal semantics (weakest preconditions) on the users in so far as they may occasionally need to appeal to semantics to justify some transformation steps. Finally, it is difficult to use in practice because it deals mostly at the level of commands and has almost nothing to say about manipulating expressions.

More recently, there have been calculi for functional programming. Those in the Squiggol family (e.g. [Bir84, Mee86, BdM97]) aim to derive efficient algorithms from equivalent, inefficient, but obviously correct algorithms.

Other researchers have explored specification languages, often adapting the concepts of The Refinement Calculus to functional languages (e.g. [NH93, Lee93, War94,

* Supported by the Leverhulme Trust

Bun97]). These facilitate the derivation of algorithmic programs, usually more economically than corresponding imperative derivations.

We present a refinement calculus for expressions that imitates in expressions the specification features that The Refinement Calculus has in commands. The programming sublanguage is a simple strict functional language. The calculus is equipped with a logic that deals adequately with both undefinedness (e.g. caused by nontermination) and nondeterminacy.

2 The calculus

We aim to construct a refinement calculus by extending a logic with features that capture specifications and programs. Since programs may be *undefined*, for instance because they don't terminate, and since specification usually are *nondetermined*, that is, they allow more than one implementation, we choose a logic that accommodates undefinedness and nondeterminacy. Such a logic, called E4, has been constructed in [Mor96, MB97]. It is called E4 because it is tuned for equational reasoning, and has four truth values. Its main connectives are equivalence (\equiv) and implication (\Rightarrow). Most theorems familiar from classical two-valued logic continue to hold, but of course not all. However, in reading this paper, it is enough to know that E4 behaves exactly like classical logic when all arguments of the logical connectives are determined — which they will be in this paper.

The main symbol used to reason about specifications and programs is the refinement relation. Expression E is refined by expression F , and we write $E \sqsubseteq F$, if a customer asking for an implementation of E would be happy when given an implementation of F . Equivalence (\equiv) is simply the antisymmetric closure of refinement. Further, expressions are classified according to whether they are defined (we write τE if E is defined) and determined (we write ΔE). We use capital letters for expressions, and small letters for variables (values). The types are ground types, functions, and pairs. We assume all expressions we write are typed. We mark propositions by "axm" if they are axioms, and "thm" if they are theorems.

$$\begin{array}{lcl}
 \text{axm } 0 & \tau E & \vee (\forall x. E \sqsubseteq x) \\
 \text{axm } \equiv & E \equiv F & \equiv (\tau E \equiv \tau F) \wedge (\forall x. (E \sqsubseteq x) \equiv (F \sqsubseteq x)) \\
 \text{axm } \sqsubseteq & E \sqsubseteq F & \equiv (\tau E \Rightarrow \tau F) \wedge (\forall x. (E \sqsubseteq x) \Leftarrow (F \sqsubseteq x))
 \end{array}$$

Definedness and refinement by a value will be the two basic building blocks for axiomatizing the specification language constructs. Axiom 0 describes how they relate, namely, that undefined expressions are refined by every value (of the appropriate type). The converse is not true: there are expressions that are defined, and still refined by every value. The next two axioms describe the basic relations of expressions: equivalence and refinement. They are mapped to equivalence and implication of propositions, via the building blocks of definedness and refinement by values. Immediately we can deduce that they inherit the properties of equivalence and implication: So equivalence on expressions is indeed an equivalence relation, and refinement is a partial order, antisymmetric with respect to equivalence. An easy proof will show undefined expressions are equivalent, so we introduce a symbol for an undefined expression (\perp).

Next, we'll extend the logic E4 by the constructs of a specification language. Each language construct is axiomatised by two axioms: The first describes when the construct is defined, and the second describes what values refine it. This strategy of axiomatization mimics denotational semantics closely, and thereby increases our confidence in completeness of the calculus. First, here are axioms for demonic choice in binary and quantified form:

$$\begin{array}{llll}
\text{axm} & \sqcap\tau & \tau(E\sqcap F) & \equiv \tau E \wedge \tau F \\
\text{axm} & \sqcap\sqsubseteq & E\sqcap F \sqsubseteq x & \equiv (E\sqsubseteq x)\vee(F\sqsubseteq x) \\
\text{axm} & \sqcap\tau & \tau(\sqcap x.E) & \equiv (\forall x.\tau E) \\
\text{axm} & \sqcap\sqsubseteq & (\sqcap x.E)\sqsubseteq y & \equiv (\exists x.E\sqsubseteq y).
\end{array}$$

$E\sqcap F$ yields an outcome of E or of F , and $\sqcap x.E$ yields an outcome of E with x instantiated by an arbitrary value (of the appropriate type). We deduce that reducing nondeterminacy is refining. Choice is demonic since $E\sqcap\perp$ is itself undefined. Demonic choice is the greatest lower bound operation with respect to refinement, that is, $(X\sqsubseteq E)\wedge(X\sqsubseteq F) \equiv (X \sqsubseteq E\sqcap F)$ and its quantified form. Furthermore $(E\sqsubseteq F)\equiv(E\sqcap F \equiv E)$.

In deriving a program from its specification, one of the main tasks is to move logical information from a proposition into an expression — and then find code that satisfies this logical information. We need language constructs that *bridge* between the surrounding logic and the language of expressions. One such bridge is the *assertion expression*, adapted from The Refinement Calculus. Its form is $P \succ E$, where P is a proposition, and E an arbitrary expression. The meaning is E if P is *true*, and undefined otherwise. We can think of P as a comment stating what information should be true at this point in the program, and therefore could be used to refine E . One useful shape of expression is $\lambda x.P \succ E$. We are only interested in applying this function to arguments that satisfy P . An evaluator could check the condition in an assertion expression, or just ignore it — if it was not *true*, then the expression is undefined, and therefore any outcome the evaluator delivers is acceptable, even the outcome nontermination!

The second bridge, and the most elaborate language construct is the *prescription*. It takes the form $(\square x:T|P \bullet E)$, and its meaning is informally E with x instantiated by a value of type T such that P is *true*, or undefined if none such values exist. If more than one such value exists, the prescription may be nondetermined. For example, an arbitrary divisor of 1980 is $(\square n:\text{Nat}|\text{divides } (n, 1980) \bullet n)$ and the surname of a student with maximal mark is

$$(\square s:\text{Student}|\forall t.\text{mark } t \leq \text{mark } s) \bullet \text{surname } s).$$

The outcome need not be restricted at all, for example, an arbitrary integer is specified by $(\square i:\text{Int}|\text{true} \bullet i)$. Prescriptions could be composed from more primitive language constructs. Here are the axioms governing the bridges:

$$\begin{array}{llll}
\text{axm} & \succ\tau & \tau(P \succ E) & \equiv (P \equiv \text{true}) \wedge \tau E \\
\text{axm} & \succ v. & P \succ E \sqsubseteq x & \equiv (P \neq \text{true}) \vee (E \sqsubseteq x) \\
\text{axm} & \text{pre.}\tau & \tau(\square x:T|P \bullet E) & \equiv (\exists x.P \equiv \text{true}) \wedge (\forall x.P \Rightarrow \tau E) \\
\text{axm} & \text{pre.}\sqsubseteq & (\square x:T|P \bullet E)\sqsubseteq y & \equiv (\exists x.P \equiv \text{true}) \Rightarrow (\exists x.(P \equiv \text{true}) \wedge (E \sqsubseteq y))
\end{array}$$

Two of the theorems are:

$$\begin{aligned} \text{thm inst. } P[y/x] &\Rightarrow (\Box x:T|P \bullet E) \sqsubseteq E[y/x] \\ \text{thm self } (\exists x.p \ x) &\Rightarrow p (\Box x:T|p \ x \bullet E) \end{aligned}$$

The specification language contains a programming sublanguage. This language is basically a simple strict functional programming language. We axiomatize conditional, functions, pairs, and recursive expressions.

The conditional is axiomatized by

$$\text{axm cond. } (\mathbf{if} \ P \ \mathbf{then} \ E \ \mathbf{else} \ F) \sqsubseteq G \equiv (P \Rightarrow E \sqsubseteq G) \wedge (\neg P \Rightarrow F \sqsubseteq G).$$

It follows from the axiomatization of implication in E4 that the conditional is undefined, if the condition is nondetermined.

Functions are axiomatized by the following axioms. In axiom 1, f must not be free in E .

$$\begin{aligned} \text{axm 0} \quad \forall f.f &\equiv (\lambda x.f \ x) \\ \text{axm 1} \quad \exists f.f &\equiv (\lambda x.E) \end{aligned}$$

$$\begin{aligned} \text{axm } \lambda \sqsubseteq \quad (\lambda x.E) \sqsubseteq f &\equiv (\forall x.E \sqsubseteq f \ x) \\ \text{axm } \beta \sqsubseteq \quad (\lambda x.E) \ x &\sqsubseteq E \end{aligned}$$

$$\begin{aligned} \text{axm app.}\tau \quad \tau(E \ F) &\equiv \tau E \wedge \tau F \wedge (\forall e.E \sqsubseteq e \Rightarrow (\forall f.F \sqsubseteq f \Rightarrow \tau(e \ f))) \\ \text{axm app.}\sqsubseteq \quad E \ F \sqsubseteq x &\equiv (\exists e.(E \sqsubseteq e) \wedge (\exists f.(F \sqsubseteq f) \wedge (e \ f \sqsubseteq x))) \end{aligned}$$

Functions introduce a new type built from other types. Therefore, in addition to the usual strategy of definedness and refinement by values for each new language construct, we must also describe what the values of this new type are, and how the constructors and destructors of it interact. For reasons that will be discussed later, we only allow abstractions $\lambda x.E$ where E is refinement-monotone in x . Some of the more important theorems about functions are:

$$\begin{aligned} \text{thm 19} \quad E (\Box x:T|P \bullet F) &\equiv (\Box x:T|P \bullet E \ F), \quad x \notin fvE \\ \text{thm 20} \quad (\Box x:T|P \bullet E) \ F &\equiv (\Box x:T|P \bullet E \ F), \quad x \notin fvE \\ \text{thm } \lambda \text{mon.} \quad (\forall x.E \sqsubseteq F) &\Rightarrow (\lambda x.E) \sqsubseteq (\lambda x.F) \\ \text{thm } \beta \equiv \quad \Delta F &\Rightarrow E[F/x] \equiv (\lambda x.E) \ F \\ \text{thm } \lambda/\sqcap \quad (\lambda x.E \sqcap F) &\sqsubseteq (\lambda x.E) \sqcap (\lambda x.F) \end{aligned}$$

In particular, function application is strict and distributes over nondeterminacy and over prescriptions on both sides. Further, forming abstractions and applications is monotone, so we can refine them piecewise.

Pairs are axiomatized by the six axioms below.

$$\begin{array}{ll}
\text{axm } 0 & \forall z. \exists x, y. z \equiv (x, y) \\
\text{axm } 1 & \forall x, y. \exists z. z \equiv (x, y) \\
\\
\text{axm } (\cdot)\tau & \tau(E, F) \equiv \tau E \wedge \tau F \\
\text{axm } (\cdot)\sqsubseteq & (E, F) \sqsubseteq x \equiv (\tau F \Rightarrow E \sqsubseteq \mathbf{fst}x) \wedge (\tau E \Rightarrow F \sqsubseteq \mathbf{snd}x) \\
\\
\text{axm } \mathbf{fst} \tau & \tau(\mathbf{fst} E) \equiv \tau E \\
\text{axm } \mathbf{fst} v. & \mathbf{fst} E \sqsubseteq x \equiv (\exists y. E \sqsubseteq (x, y))
\end{array}$$

Since all datatype constructors except abstractions are strict in their constituent subexpressions, recursive expressions only make sense for functions. Therefore, recursive expressions will usually be of the form $\mu f. \lambda x. E$. We axiomatize recursion as a fixpoint, and the least prefixpoint with respect to refinement. The body of the recursive expression $\mu x. E$, that is, E , must be refinement-monotone in x . This ensures fixpoints exist, and conveniently implies that the corresponding abstraction $\lambda x. E$ is acceptable.

$$\begin{array}{ll}
\text{axm } \mathbf{fix} & (\mu x. E) \equiv E[(\mu x. E)/x] \\
\text{axm } \mathbf{prefix} & E[F/x] \sqsubseteq F \Rightarrow (\mu x. E) \sqsubseteq F
\end{array}$$

The theorem $\mu\mathbf{fun}$ is the one that is generally used to introduce recursive functions.

Theorem 1 ($\mu\mathbf{fun}$) $(\lambda x. E) \sqsubseteq (\mu f. \lambda x. F)$, if $\forall x. E \sqsubseteq F[(\lambda y. (y < x) \triangleright E[y/x])/f]$, where F is monotone in f and $<$ is a well-order (no infinitely decreasing chains) on the type in question (usually integer).

□

Roughly, by finding a recursive refinement of E , we have found a recursive implementation of $\lambda x. E$. Every programmer knows the theorem well: to show a recursion (or loop) terminates, one must show that some natural number expression is reduced on each unfolding (or iteration), and that there is a lower limit (for instance 0), at which no further unfolding (or iteration) is needed.

3 Using the calculus

To give a brief taste, here's what some of the steps in specifying and deriving a very simple program may look like. Assume we need to find an integer i in $0..100$ that is mapped to the least integer by a given function g . The specification is

$$(\Box i: Z | (0 \leq i \leq 100) \wedge (\forall j. (0 \leq j \leq 100) \Rightarrow (g \ i \leq g \ j)) \bullet i).$$

We aim for a recursive function, so we first parameterize over the upper limit:

$$(\lambda x. (\Box i: Z | (0 \leq i \leq x) \wedge (\forall j. (0 \leq j \leq x) \Rightarrow (g \ i \leq g \ j)) \bullet i)) \ 100.$$

The body is trivially refined by 0 if $x=0$. Otherwise, we make use of a recursive call, decrementing the argument. We use theorem $\mu\mathbf{fun}$ with

$$E \equiv (\Box i: Z | (0 \leq i \leq x) \wedge (\forall j. (0 \leq j \leq x) \Rightarrow (g \ i \leq g \ j)) \bullet i)$$

and

$$F \equiv \text{if } x=0 \text{ then } 0 \text{ else } (\text{if } g(f(x-1)) \leq g x \text{ then } f(x-1) \text{ else } x)$$

to derive the program

$$(\mu f. \lambda x. \text{if } x=0 \text{ then } 0 \text{ else } (\text{if } g(f(x-1)) \leq g x \text{ then } f(x-1) \text{ else } x)) 100.$$

4 Design issues

The restriction on λ abstractions mentioned earlier, namely that the body should be monotone in the bound variable, must be imposed to retain consistency. That is, for $\lambda x. E$ to be well-formed, we require

$$F \sqsubseteq G \Rightarrow E[F/x] \sqsubseteq E[G/x].$$

If the bodies of λ abstractions were allowed to be not monotone in the bound variable, then the language would become inconsistent. For example, take

$$f \equiv (\lambda g. (g \equiv b)),$$

for some not flat type T with values $a \sqsubset b$. Then we would have

$$\text{false} \equiv f a \equiv f(a \sqcap b) \equiv f a \sqcap f b \equiv \text{true} \sqcap \text{false}.$$

By generalizing this argument, one could show that all expressions are equivalent! Therefore we require abstraction bodies to be monotone in the bound variable.

Ideally one would wish a calculus that is consistent and complete. We believe a model can be made by the normal techniques of denotational semantics. Demonic Smyth powerdomains would provide a model for nondeterminacy. A model along these lines for a similar (but lazy) specification language has been given in [Bun97].

In this short overview, the less practical side of the calculus has been kept hidden. There are duals to the concepts undefinedness (\perp), demonic choice (\sqcap), and assertion expressions ($P \triangleright E$). They are miracles (\sqtop), a kind of angelic choice (\sqcup), and guarded expressions ($P \rightarrow E$). A miracle is an overspecified expressions; it has no implementations. $E \sqcup F$ is implemented by the intersection of implementations of E and those of F . The guarded expression $P \rightarrow E$ is equivalent to E if P is *true*, and a miracle in all other cases. These constructs are useful for calculational techniques that communicate in small steps between logical information and the expressions under construction, and for composing more elaborate language constructs like prescriptions or conditionals from simpler ones. For example, with the added language construct **if** E **fi** which is equivalent to E if E is feasible (i.e. not miraculous), and undefined otherwise, we can compose the prescription ($\sqcap x:T | P \bullet E$) as **if** $\sqcap x:T. P \rightarrow E$ **fi**, and the conditional **if** P **then** E **else** F as **if** $P \rightarrow E \sqcap \neg P \rightarrow F$ **fi**.

The disadvantage of using these hidden language constructs is that they introduce miracles. A miracle is an expression that delivers exactly the outcome you want, even if that is impossible, for example, the integer i satisfying *false*. Of course miracles are not implementable. They are problematic for two reasons. Firstly, they disturb the use of

the refinement relation: In developing a program, we can always refine the program and still be on the right track. But if there are possibly miraculous expressions, at each step we have to add a feasibility-check. Have we refined too much? Are there still possible implementations? Clearly such seemingly extra work is undesirable. However, even if there are no possibly miraculous expressions in the language, the proof work of these checks still shows up — just in different places, e.g. in the assumption of the axiom giving refinement by a value of a prescription. The second reason why miracles are undesirable is that since we don't distinguish between propositions and boolean expressions, the number of truth "values" rises to a disconcerting *five*: \perp , *true* \sqcap *false*, *true*, *false*, and the miraculous \top . However, miracles can only arise from guarded expressions and \sqcup -nondeterminacy — therefore a feasibility check can be reduced to a syntactic check.

5 Future work

The immediate further work is to axiomatize more data types, say along the lines of the algebraic data types of Haskell.

Our reasons for making application and data type construction *strict* are mainly conventional: All widely-used programming languages are strict, and most of the work about refinement calculi so far has dealt with strict languages. However, lazy languages have since become a viable alternative, and calculi for them are being explored (e.g. [Bun97]). Certainly, the axioms given above put the spotlight on undefinedness — possibly more than it deserves. Maybe a lazy calculus would integrate undefinedness more smoothly. If variables could be values, or undefined (that is, τx is a contingency), then within a lazy calculus a strict version of λ abstraction could be given by $\lambda x.\tau x \succ E$, and similarly for the other quantifiers. It remains to be seen whether such an extended calculus works out nicer calculationally.

A further step would be to allow variables to range over expressions, defined or undefined, determined or nondetermined, feasible or miraculous. Such a calculus would axiomatize binders like $\lambda X.E$ where E contains the expression-variable X . The lazy binders could be recovered by

$$\lambda X.(\neg\tau X \vee \Delta X) \succ E$$

and the strict abstraction used in the body of this paper would be

$$\lambda X.\Delta X \succ E.$$

Such a calculus would provide the beautiful "disentanglement" theorem

$$(E \sqsubseteq F) \equiv (\forall X. E \sqsubseteq X \Leftarrow F \sqsubseteq X),$$

whereas the similar

$$(E \sqsubseteq F) \equiv (\forall x. E \sqsubseteq x \Leftarrow F \sqsubseteq x)$$

is not valid — it doesn't distinguish between undefinedness and maximal nondeterminacy.

Originally, this line of research grew out of a dissatisfaction with the treatment of expressions (the right-hand sides of assignments) in The Refinement Calculus. There expressions were always assumed to be determined, and the expression language is not much discussed, but would certainly not provide specificational constructs or recursion. We aim to merge The Refinement Calculus and Expression Refinement by allowing our rich expression language to occur as right hand sides of assignments, and by providing a languages construct that encapsulates a command into an expression — along the lines of the imperative monad-comprehensions in Haskell.

References

- [Bac80] R.-J. R. Back. Correctness preserving program refinements: Proof theory and Applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.
- [BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.
- [Bir84] R. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, 6(4), October 1984.
- [Bun97] A. Bunkenburg. *Expression Refinement*. PhD thesis, Computing Science Department, University of Glasgow, 1997.
- [Lee93] René Leermakers. *The Functional Treatment of Parsing*. Luwer Academic Publishers, 1993.
- [MB97] J. M. Morris and A. Bunkenburg. Undefinedness and nondeterminacy in program proofs. Submitted, 1997.
- [Mee86] L. Meertens. Algorithmics - Towards Programming as a Mathematical Activity. *Mathematics and Computer Science*, 1, 1986. CWI Monographs (J. W. de Bakker, M. Hazewinkel, J. K. Lenstra, eds.) North Holland, Puhl. Co.
- [Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287 – 306, 1987.
- [Mor88] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10:403 – 419, 1988.
- [Mor94] C. Morgan. *Programming from Specifications*. Prentice Hall, 2nd edition, 1994.
- [Mor96] J. M. Morris. Reasoning equationally in the presence of undefinedness. *Science of Computer programming*, 1996. Submitted for publication.
- [NH93] T. S. Norvell and E. C. R. Hehner. Logical specifications for functional programs. In *Proceedings of the Second International Conference on Mathematics of Program Construction, Oxford, 29 June - 3 July 1992*, volume

669 of *Lecture Notes in Computer Science*, pages 269–290. Springer Verlag, 1993.

[War94] N. Ward. *A Refinement Calculus for Nondeterministic Expressions*. PhD thesis, University of Queensland, 1994.