Well-Going Programs Can Be Typed

Stefan Kahrs

University of Kent at Canterbury
Computing Laboratory
Canterbury CT2 7NF
United Kingdom
email: smk@ukc.ac.uk

Abstract. Our idiomatically objectionable title is a pun on Milner's "well-typed programs do not go wrong" — because we provide a completeness result for type-checking rather than a soundness result.
We show that the well-behaved functions of untyped PCF are already expressible in typed PCF: any equivalence class of the partial logical equivalence generated from the flat natural numbers in the model given by PCF's operational semantics is inhabited by a well-typed term.

1 Introduction

The functional programming language PCF [13, 10, 5] can be regarded as the kernel of all typed functional programming languages. It is expressive enough to closely mirror the operational or denotational semantics of the full languages and small enough to make a serious study of these semantics feasible.

The subject of this paper is the operational semantics of PCF and the question whether the type system of PCF prevents us from writing meaningful PCF programs. Of course, there are programs which syntactically fail to type-check and behave in exactly the same way as well-typed programs (hint: think about dead code), but it is not *a priori* clear whether there are others that also fail to type-check but exceed the expressive power of well-typed programs.

However, we do not want to give up soundness as a price for occasional wizardry. In other words, such programs should never exhibit elementary soundness violations such as using a number as a function or a function as a number (in an evaluation context). The absence of such additional functions would mean that PCF is *complete*, in the sense of [6, 7].

This notion of completeness is indeed a dual to soundness — based on the partial logical equivalence we get by relating every value[1] of the flat natural numbers to itself. PCF's type system is sound since every well-typed term at type σ inhabits one of the equivalence classes, and we show that it is complete by demonstrating that every equivalence class is inhabited by a well-typed term. One consequence of this is that quotienting this term model by the partial equivalence gives us a fully abstract model of typed PCF.

[1] This is a slight simplification as we are actually dealing with a term model and have to close the relation under reduction.

We show completeness by writing an interpreter for untyped PCF in typed PCF — or rather, one interpreter for each PCF type. There is a technical difficulty here: when we interpret an untyped term at *a function type* $\alpha \to \beta$ then the interpretation has to incorporate values of type α; moreover, if α is itself a function type then it becomes impossible to syntactify these values — function types are abstract data types in a functional programming language. The solution to this problem is a combination of symbolic evaluation and evaluation in Dana Scott's model D_∞ [1]. We cannot implement full D_∞ in a PCF program, because that would involve infinitely many types. We can implement a finite fragment though, and we can statically determine (from the type into which we want to interpret) which fragment suffices for this purpose.

The aim of this paper is to show that every untyped PCF program p that *behaves well* at type σ is equivalent to well-typed program p' at that type. Technically, p "behaving well" means that $p \stackrel{\sigma}{\approx} p$, and p being "equivalent" to p' means $p \stackrel{\sigma}{\approx} p'$, where $\stackrel{\sigma}{\approx}$ is the partial equivalence generated from the flat natural numbers in the term model. Notice that this is a rather strong notion of well-behaviour: p not only fails to exhibit soundness violations, it also fails to break extensionality.

2 Implementing PCF

We assume familiarity with functional programming and the simply typed λ-calculus. PCF itself is a very simple typed functional programming language, which has natural numbers as its base type, supports general recursion and the simply typed λ-calculus.

As concrete syntax for writing PCF programs we choose Haskell [4] and allow ourselves the liberty to exploit its much richer syntax, for example we use general (and mutual) recursion instead of fixpoint operators and its class system. The source code of the implementation can be found at
`http://www.cs.ukc.ac.uk/~smk/PCF/Untyped_PCF.html`.

There are several reasons for not using PCF itself. First, there is the pragmatic one that too many things would have to be encoded, making the programs much longer and more difficult to read. Moreover, the structuring mechanism of Haskell (we exploit classes) can be used to emphasize general ideas. Most importantly, it also means that we can write a single schematic (and thus finite!) implementation instead of a separate one for each type — the techniques used to achieve that are similar to the ones describe by McBride in [9]; this necessarily involves the use of some compiler-supported language extensions such as *multi-parameter type classes* and *functional dependencies*, because we need to make some computations at type level.

However, these computations at type-level are static and serve to stress that the implementation is schematic. The types we use remain pure PCF types.

2.1 Symbolic Evaluation

Part of the implementation is entirely syntactic, i.e. it operates directly on the abstract syntax and does not involve imported values.

```
data Exp = App Exp Exp | Lam Exp | Var Int |
           Zero | Succ Exp | Pred Exp | If0 Exp Exp Exp |
           Fix Exp
```

The above defines (the abstract syntax of) open PCF expressions as a Haskell datatype; we represent variable binding via de Bruijn indices [2]. For our target language of typed PCF the type `Exp` is no more than syntactic sugar for ι.

The first line of the type definition shows the λ-calculus fragment of PCF, the second all the operations dealing with natural numbers; the final is a fixpoint operator — which the *untyped* language could actually do without, because its λ-calculus fragment is powerful enough to express it.

We need a notion of substitution for these terms, but it is completely routine — see appendix A.

The goal of the implementation is to stay syntactic as much as possible, and only resort to type casting once we use a variable in an evaluation context that demands its value. Therefore, we first define a type for normal forms (another piece of syntactic sugar for ι) and a normalizing function that reduces expressions to their normal forms. Although this works in an untyped setting, certain elementary soundness violations can occur — this is the reason why the result type of normalisation is `Maybe NForm`: a `Nothing` result indicates the occurrences of such an elementary soundness violation during expression normalisation.

```
data NForm =
    Hnf Int [Exp] | Abst Exp |
    Con Int | Pr NForm | Su NForm | Test NForm Exp Exp

normalize :: Exp -> Maybe NForm
normalize e = norm [] e

norm :: [Exp] -> Exp -> Maybe NForm
norm xs (Var k) = return (Hnf k xs)
norm [] Zero = return (Con 0)
norm _ Zero = Nothing
norm xs (Succ t) = norm xs t >>= su
norm xs (Pred t) = norm xs t >>= pr
norm xs (Fix t) = norm xs (subst0 (Fix t) t)
norm xs (App f a) = norm (a:xs) f
norm (x:xs) (Lam t) = norm xs (subst0 x t)
norm [] (Lam t) = return(Abst t)
norm (x:xs) (If0 c t e) =
    norm xs (If0 c (App t x)(App e x))
norm [] (If0 c t e) =
```

```
        normalize c >>= (\nf->
            case nf of
                Con 0 -> norm [] t
                Con k -> norm [] e
                Abst _ -> Nothing
                z -> return (Test z t e))
```

For non-Haskellists it should be explained that `>>=` is function application in the Kleisli category of a monad, in this case the lifting monad. For example, the function `su` (see appendix B) attempts to add 1 to a normal form, but we only use it here if the evaluation of its meant-to-be argument gave a normal form and not a type violation.

2.2 Interpreting Normal Forms at a Type

We are going to evaluate normal forms in an environment with bound variables of certain type. We will use only certain types for this purpose. More specifically, these types need to be instances of the following classes:

```
class IntEmbed a where
    int2a :: Int -> a
    a2int :: a -> Int

class SelfEmbed a where
    apply :: a -> a -> a
    lambd :: (a->a) -> a
```

Implicitly[2] we also require that these two maps form in each case retractions. We come to that later.

Concerning the relationship between this code and its PCF equivalent, it should be clear that the use of Haskell's class mechanism improves the presentation, but it is actually unnecessary: the four components of these two classes can always be passed around as additional parameters. Also, the following functions have as one of their parameters a list with component type `a` — we do not have lists in PCF but we can easily model them, e.g. as an integer (the length) plus a function of type $\iota \to$ `a` for the list indexing.

This list argument incidentally serves as a stack, giving meaning to the free variables of the expressions (or normal forms). The de Bruijn indices are relative addresses into this stack.

```
evalNF :: (SelfEmbed a,IntEmbed a) => [a] -> NForm -> a
evalNF xs (Hnf n es) = applyargs (xs!!n) [evalExp xs e | e<-es]
evalNF xs (Abst e) = lambd (\a -> evalExp (a:xs) e)
evalNF xs (Con n) = int2a n
```

[2] This is not directly enforceable in Haskell, but we achieve it by only providing instances that match or preserve the properties.

```
evalNF xs (Pr e) = liftInt prednat (eval xs e)
evalNF xs (Su e) = liftInt (+1) (eval xs e)
evalNF xs (Test c t e) =
    case a2int (evalNF xs c) of
        0 -> evalExp xs t
        _ -> evalExp xs e

applyargs :: SelfEmbed a => a -> [a] -> a
applyargs v [] = v
applyargs v (e:es) = applyargs (apply v e) es

evalExp :: (SelfEmbed a,IntEmbed a) => [a] -> Exp -> a
evalExp xs e = evalNF xs (unwrap(normalize e))

unwrap :: Maybe a -> a
unwrap (Just x) = x
unwrap Nothing = error "type violation"
```

The most interesting part of this is how we deal with abstractions and applications, because these are the two cases which involve the self-embedding. The first case deals with the application of a variable to a sequence of expressions. We fetch the value of the variable from the stack and subsequently apply it to the evaluations results of the expressions.

Notice that although we pass *evaluated* expressions to the application this does not mean that the application mechanism is using applicative evaluation order. If our implementation language uses normal evaluation order then these evaluations remain delayed until their values are requested. However, to achieve that preservation of evaluation order we had to employ the above `unwrap` function instead of evaluating in the lifting monad. The code of some auxiliary functions (e.g. `liftInt`) can be found in appendix B.

2.3 Interpreting at a Type

Our goal is to interpret (untyped) expressions at an arbitrary type. This effectively means to have one function $\texttt{interpret}_\sigma$ of type $\texttt{Exp} \rightarrow \sigma$ for every PCF type σ. In Haskell we can express this using a class:

```
class Run t where interpret :: Exp -> t
```

Clearly, we can instantiate this class at `Int`.

```
instance Run Int where interpret = evalExp []
```

However, things get a bit more complicated when we want to interpret function types. As we shall see, this requires certain computations on types, and Haskell 98 is too restrictive to aid in that task. Still, the Glasgow Haskell Compiler `GHC` supports several language extensions which *are* sufficient for our purposes.

Be aware that we only need type computations in order to provide a *single, schematic* implementation of `interpret`; for the completeness goal in PCF it suffices to provide an implementation for $\texttt{interpret}_\sigma$ which can be obtained from instantiating the schema with σ.

The first of the language extensions we employ are multi-parameter classes [12].

```
instance (LUB a b c,Full c d) => Run(a->b) where
    interpret e a = (p1.p2)(evalExp [(e2.e1) a] (expand e))
                    where e1 = embed :: a->c
                          e2 = embed :: c->d
                          p2 = project :: d->c
                          p1 = project :: c->b
```

This may look rather bewildering at first. The essential idea is the following:

- we try to interpret an expression as a function of type $\alpha \to \beta$,
- we do this by waiting for the argument, pushing it on the stack and running our interpreter,
- except that the interpreter does not work with arbitrary types, only with the approximants of D_∞,
- so we need to find a sufficiently large one of those (which is `d`),
- lift the argument `a` via an embedding into `d` and project the result to the expected type `b`.

The function `expand` is defined in appendix A. Essentially, it is like an η-expansion without the λ, where the meaning of the free variable 0 is now provided by the stack. The indicated computations on types appear in the premise of the class instance.

The class `LUB` is provided in section 3; it exploits another extension feature, functional dependencies, which in this case guarantee that the first two parameters uniquely determine the third. Without that guarantee the use of the free type variable `c` in the premise of the class instance would not be permitted. A similar exploitation is made for class `Full`. The type annotation inside the right-hand side are necessary to disambiguate which embedding we use. `GHC`[3] keeps the type variables from an instance declaration free in the context in which its members are type-checked; this means that we are referring to the same type `d` here.

In general, it will not be the case that the type `c` we get from the least upper bound computation is suitable for evaluation. For example, the type $(\iota \to \iota \to \iota) \to \iota$ would not be. Therefore we have to find a type which is, and this is the purpose of the class constraint `Full c d`. Class `Full` is defined in such a way

[3] The Haskell interpreter `Hugs` which generally supports very similar language extensions to `GHC` does not do that, e.g. `d` would be bound more locally — which is not what we want here. As a result, it is awkward if not impossible to replicate this implementation in Hugs.

that `d` will be the smallest of the D_i domains of D_∞ into which both `a` and `b` can be faithfully embedded.

This provides us now with an interpretation of expressions at *any* PCF type, provided we can find for any PCF type `a` a (unique) type `b` for which we have `Full a b`.

3 Retractions

Just as the D_∞ model, our implementation makes extensive use of retractions. We give retractions an extra section, partly because of this extensive use and partly because they play a central role when we want to reason about the implementation. However, most of this is fairly routine and follows the typical patterns one can find e.g. in chapters 11 and 12 of [3].

Definition 1. *A* retraction *between two objects A and B in a category is a pair of morphisms $\phi : A \to B$ and $\psi : B \to A$ such that $\psi \circ \phi = id_A$. We call ϕ the* embedding *and ψ the* projection.

We replicate that definition (without the condition) in Haskell, keeping in mind that all instances of the class should satisfy the condition. This requires multi-parameter classes. Of course, the PCF implementation cannot use such a schematic presentation and we have to provide these functions by hand; as a result, the PCF implementation increases in size as we move up the type hierarchy.

```
class Retraction a b where
    embed :: a->b
    project :: b->a
```

There is an added bit of security in Haskell which we get for free: although there are typically many possible retractions between two types (provided there is one at all), the compiler enforces that we only have one instance of the class for the given types.

We can give retractions between base type ι and any PCF type. For technical reasons (to pacify the `GHC` compiler), the code has to single them out.

```
class IntEmbed a where
    a2int :: a -> Int
    int2a :: Int -> a

instance IntEmbed Int where
    a2int = id
    int2a = id

instance (IntEmbed a,IntEmbed b) => IntEmbed (a->b) where
    a2int f = a2int (f (error "function is not a number"))
    int2a n _ = int2a n
```

```
instance IntEmbed a => Retraction Int a where
    embed = int2a
    project = a2int
```

The call to `error` is rather convenient here in Haskell, since it is triggered only if the function from which we retract touches its argument. In PCF we would have to resort to an ordinary non-termination instead.

In the categorical setting it should be obvious that functors preserve retractions: $F(\psi)\circ F(\phi) = F(\psi\circ\phi) = F(id) = id$; if F is contravariant then embedding and projection swap places. We use this to lift retractions to function types, e.g. defining the function space functor.

```
(-->) :: (a->b) -> (c->d) -> (b->c) -> (a->d)
f --> g = \ h -> g . h . f

instance (Retraction a c,Retraction b d) =>
    Retraction (a->b) (c->d) where
        embed = project --> embed
        project = embed --> project
```

This already sets up all the retractions between PCF types. Retractions generally define a preorder between the objects in a category, where $a \leq b$ iff there is a retraction from a to b. As previously indicated, we are interested in the least upper bounds of this preorder.

```
class (Retraction a c, Retraction b c) => LUB a b c | a b -> c
```

The bit at the end is a "functional dependency" which declares that types `a` and `b` uniquely determine `c`. This is enforced by the compiler.

Notice that the class has no members of its own. It is solely a device to convince the system that certain retractions exist.

```
instance LUB Int Int Int
instance (IntEmbed b,IntEmbed a,Retraction a a,Retraction b b) =>
    LUB Int (a->b) (a->b)
instance (IntEmbed a,IntEmbed b,Retraction a a,Retraction b b) =>
    LUB (a->b) Int (a->b)
instance (LUB a1 b1 c1, LUB a2 b2 c2) =>
    LUB (a1->a2) (b1->b2) (c1->c2)
```

Lemma 1. *All PCF types have least upper bounds in the retraction preorder.*

The proof is a trivial induction on the type structure, using exactly the pattern from the class instances.

We also need to single out types which are suitable for self-embeddings.

```
class SelfEmbed a
    where apply :: a -> a -> a
          lambd :: (a->a) -> a
```

For this purpose, we single out the finite approximations of D_∞:

```
instance SelfEmbed Int
    where apply n _ = n
          lambd f = f (error "use of function as number")

instance SelfEmbed a => SelfEmbed (a->a)
    where apply = lambd --> apply
          lambd = apply --> lambd
```

Lemma 2. *Every PCF type can be embedded into a PCF type that supports self-embedding.*

To prove the lemma, we need instances `Full a b` for every type `a`, for the following class `Full`, again a class without members.

```
class (Retraction a b,IntEmbed b,SelfEmbed b) => Full a b | a -> b
```

These instances are generated recursively as follows:

```
instance Full Int Int
instance (Full a x, Full b y, LUB x y c,
    Retraction a c, Retraction b c, IntEmbed c, SelfEmbed c) =>
        Full (a->b) (c->c)
```

The instance for the function type has a few more conditions than one might expect. The reason is that it is (i) impossible to tell `GHC` that the retraction relation is transitive and (ii) we do not know (in general) that the least upper bound of two self-embedding types has the required properties. For our purposes this is true, because the least upper bound of D_k and D_p is simply $D_{\max(k,p)}$.

4 The Term Model

The implementation gives us a function to interpret expressions at some type. But is its behaviour really the same as that of the original expression? In other words, we are trying to compare a PCF expression e at type σ with the PCF expression $\texttt{interpret}_\sigma(\ulcorner e \urcorner)$, where $\ulcorner e \urcorner$ is the Gödel number of e.

We need a bit of machinery to turn this into a meaningful question. First, recall from [11] that a *typed applicative structure* is a tuple

$$\langle \{A^\sigma\}, \{\mathbf{App}^{\sigma,\tau}\} \rangle$$

of families of sets and application maps indexed by type expressions. Compared to [11] I omitted the constants, i.e. this is viewed w.r.t. the empty signature.

Our untyped expressions provide us with a very simple typed applicative structure: A^σ is `Exp`, regardless of type, and **App** is just the `App` operation on `Exp`.

From this model we want to carve out the well-behaved expressions at the various types. We do this by defining a *partial equivalence relation*, i.e. a symmetric and transitive relation. Moreover, we define a family of such relations $\overset{\sigma}{\approx}$, one for each type σ — a *logical* relation. At base type we define:

$$a \overset{\iota}{\approx} b \iff \begin{cases} \exists k.\, \texttt{normalize}\ a = \texttt{Just(Con}\ k) = \texttt{normalize}\ b, \text{or} \\ (\texttt{normalize}\ a)\!\uparrow \wedge\, (\texttt{normalize}\ b)\!\uparrow \end{cases}$$

Here, $p\uparrow$ means that the evaluation of the expression p fails to terminate. In other words, at base type we relate all expressions to other expressions that compute the same number, and we also relate expressions that fail to terminate to each other. Not covered by this relation are the expressions which either evaluate to `Nothing` (a run-time type violation) or which evaluate to a normal form which is not a number.

As usual, this is lifted in a canonical way to function types:

$$a \overset{\sigma\to\tau}{\approx} b \iff \forall d, e.\, d \overset{\sigma}{\approx} e \Rightarrow \texttt{App}\ a\ d \overset{\tau}{\approx} \texttt{App}\ b\ e$$

An expression e is now viewed to be well-behaved at type σ iff it is related to itself at that type: $e \overset{\sigma}{\approx} e$.

Theorem 1. *For all types σ and all PCF expressions e:*

$$e \overset{\sigma}{\approx} e \Rightarrow e \overset{\sigma}{\approx} \texttt{interpret}_\sigma(\ulcorner e \urcorner)$$

where $\ulcorner e \urcorner$ is the Gödel-number of e, i.e. the corresponding value at type `Exp`*.*

This implies that every equivalence class of the partial equivalence is inhabited by a well-typed term, simply because $\texttt{interpret}_\sigma(\ulcorner e \urcorner)$ is well-typed by construction.

The proof works by induction on the type structure, here is a sketch:

– At base type we get this very easily: every integer is expressible in PCF, and we do have well-typed non-terminating programs at this type as well.
– Suppose we have $t \overset{\sigma\to\tau}{\approx} t$ and arbitrary $u \overset{\sigma}{\approx} u'$. We need to show $t\ u \overset{\tau}{\approx} \texttt{interpret}_{\sigma\to\tau}(\ulcorner t \urcorner) u'$. Exploiting (i) the induction hypotheses on both σ and τ, (ii) the PER properties, and (iii) the fact that t and $\texttt{interpret}_{\sigma\to\tau}(\ulcorner t \urcorner)$ both inhabit $\overset{\sigma\to\tau}{\approx}$ we can reduce this problem to showing the following:

$$\texttt{interp}_{\sigma\to\tau}(\ulcorner t \urcorner)\,(\texttt{interp}_\sigma(\ulcorner u \urcorner)) \overset{\tau}{\approx} \texttt{interp}_\tau(\ulcorner t\ \texttt{interp}_\sigma(\ulcorner u \urcorner) \urcorner).$$

However, the only difference between interpretation on the left and the right is merely that the left will keep the term $\texttt{interp}_\sigma(\ulcorner u \urcorner)$ in the environment while the right has it in-lined.

The equivalence between evaluation in an environment and evaluation after substitution is normally explained by the substitution lemma for PCF. This does not quite apply here since the term t may not itself be well-typed, i.e. we may not have $\vdash t : \sigma \rightarrow \tau$. However, syntactic well-typedness can easily be replaced (for the purposes of a substitution lemma) by a semantic property, here that t inhabits $\overset{\sigma\rightarrow\tau}{\approx}$.

5 Related Work

In [7] we showed the *absence* of completeness for the programming language Standard ML. Although it would be possible to mirror the approach from this paper and write an interpreter for untyped SML within typed SML, it is not always possible to import/export values between these worlds and it is this fact that makes all the difference.

Marz and others [8] also linked typed PCF to an untyped language they called $\mathcal{L}$, but they are interpreting typed PCF via $\mathcal{L}$ while we are doing essentially the opposite. Their paper is lacking (but conjecturing) a "universality" result, and this is essentially the completeness of our paper. There is one major difference between $\mathcal{L}$ and untyped PCF: $\mathcal{L}$ has a unary operator "`where`" that enables $\mathcal{L}$ programs to distinguish between numbers and functions, and that goes beyond the capabilities of untyped PCF. Is typed PCF complete w.r.t. $\mathcal{L}$? This is not as clear-cut as one might think, and the reason is the following: the problematic `where` operator allows one to distinguish at a function type (say $\iota \rightarrow \iota$) between $\perp$ and $\lambda x.\perp$ — where $\perp$ is some suitable non-terminating expression. Typed PCF cannot make this distinction since these two terms are logically related, and thus we cannot implement this operator in its full glory in typed PCF.

6 Conclusion

We have shown how all "sound" values of PCF types in the untyped term model can be expressed by typed PCF expressions. It also follows from this result that if we quotient the term model of untyped PCF by the indicated partial equivalence we get fully abstract model of typed PCF: it is a model because PCF typing is sound, and it is fully abstract because every equivalence class is inhabited by a typed term.

The reason a self-implementation in this style does not work for Standard ML with "nested" recursive datatypes [7] is that the argument type of a constructor for nested datatypes can be "larger" than its result type and as a result we would not get away with having a *finite* number of class instances. A more interesting question is whether such a self-implementation would be possible in Haskell.

A Substitution Code

```
subst :: Exp -> (Int->Exp) -> Exp
subst (Var n) f = f n
subst (App t u) f = App (subst t f) (subst u f)
subst (Lam t) f = Lam (subst t (Var 0 <: f))
subst (Fix t) f = Fix (subst t (Var 0 <: f))
subst Zero f = Zero
subst (Succ t) f = Succ (subst t f)
subst (Pred t) f = Pred (subst t f)
subst (If0 c t e) f =
    If0 (subst c f) (subst t f) (subst e f)

(<:) :: Exp -> (Int->Exp) -> Int -> Exp
t <: e = aux
         where aux 0 = t
               aux n = lift (e(n-1))

lift :: Exp -> Exp
lift t = subst t (Var . (+1))

subst0 :: Exp -> Exp -> Exp
subst0 t u = subst u (t <: (Var . (subtract 1)))

expand :: Exp -> Exp
expand e = App(lift e)(Var 0)
```

Remark: the described functions `subst`, `lift`, and `<:` are mutually recursive but it is clear that the mutual recursion could be resolved.

B Auxiliary Functions on Normal Forms

```
su :: NForm -> Maybe NForm
su (Con x) = return (Con (x+1))
su (Abst _) = Nothing
su x = return (Su x)

pr :: NForm -> Maybe NForm
pr (Con x) = return (Con (prednat x))
pr (Abst _) = Nothing
pr x = return (Pr x)

prednat :: Int -> Int
prednat n = if n==0 then 0 else (n-1)
```

```
liftInt :: IntEmbed a => (Int->Int) -> (a->a)
liftInt = a2int --> int2a
```

References

1. Hendrik P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics.* North-Holland, 1984.
2. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.
3. J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.
4. P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell, a Non-strict, Purely Functional Language version 1.3. Technical report, University of Glasgow, 1996.
5. Achim Jung and Allen Stoughton. Studying the fully abstract model of PCF within its continuous function model. In *Typed Lambda Calculi and Applications*, 1993. LNCS 664.
6. Stefan Kahrs. About the completeness of type systems. In *Proceedings ESSLLI Workshop on Observational Equivalence and Logical Equivalence*, 1996.
7. Stefan Kahrs. Limits of ML-definability. In *Proceedings of PLILP'96*, pages 17–31, 1996. LNCS 1140.
8. Michael Marz, Alexander Rohr, and Thomas Streicher. Full abstraction and universality via realisability. In *LICS'99*, pages 174–182, 1999.
9. Conor McBride. Faking it. *Journa of Functional Programming*, 12(4,5):375–392, July 2002.
10. Robin Milner. Fully abstract models of types λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.
11. J.C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 8, pages 365–458. Elsevier, 1990.
12. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop 1997*, pages 1–16, 1997.
13. Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.