# Signed and sealed

Philip Wadler Jeremy Yallop
University of Edinburgh

## Abstract

Language constructs for defining abstract types commonly come in two varieties: those that add and remove seals dynamically as values cross the abstraction boundary, and those that define the boundary statically using a type signature. Abstract types in dynamically-typed languages are generally defined using seals whereas statically-typed languages more typically use a signature; two prominent exceptions are Haskell, which uses seals, and Standard ML, which provides for both styles.

We show that the two styles are interconvertible, and give a proof based on Pitts' formulation of relational parametricity. In the light of this equivalence we revisit the decision to use seals for abstract types in Haskell and describe a library which extends Haskell with a construct for defining abstract types using signatures by a translation which inserts seals as necessary.

## 1. Introduction

**Hiding in plain view** Manufacturers sometimes try to prevent customers investigating the inner workings of their products. There are two common approaches. The manufacturer may seal the product, offering no way of breaking the seal. In this case the customer is free to investigate the product without risk of censure, but physically prevented from accessing its inner workings. Alternatively, the manufacturer may require that customers assent to a license that proscribes investigation. In this case the customer is legally obliged not to pry, even though there is no physical barrier to investigation. The goal, whether achieved by locks or laws, is the same: to allow access to a subset of the product's functionality while protecting the underlying mechanism from external access.

We find a similar dichotomy when we turn to abstract types, the topic of our paper. An abstract type definition creates a boundary between the part of a program that *defines* a type (the "manufacturer"), which acts on its representation, and the part that *uses* the type (the "customer"), which acts on its interface. There are two common mechanisms for defining this abstraction boundary. The first involves tagging values of the type as they cross from the section where the abstract type is defined to the section where it is used. For example, in Standard ML we might define a type of complex numbers as follows.

```
local
  datatype complex = Complex of (real × real)
in
  fun make (x,y) = Complex (x,y)
  fun real (Complex (x,y)) = x
  fun imag (Complex (x,y)) = y
  fun conj (Complex (x,y)) = Complex (x, ~y)
  fun plus (Complex (u,v), Complex (x,y))
        = Complex (u+x, v+y)
end
```

This introduces a new type, *complex*, defined in terms of a pair of *real*s. The *Complex* data constructor can be used to construct and deconstruct values of the new type, but the *local* keyword delimits the scope of the *Complex* data constructor to the section of the program between **in** and **end**; in other parts of the program such values are "sealed", and cannot be deconstructed. Values of type *complex* are isomorphic to pairs of reals, but have a distinct representation in the semantics. We dub this style of abstraction **sealing**.

The second style of abstraction uses a type signature to distinguish between values of the abstract type and values of the type with which it is implemented. Again, using Standard ML we might choose to define a type of complex numbers as follows.

```
structure Complex =
struct
  type complex = real × real
  fun make (x,y) = (x,y)
  fun real (x,y) = x
  fun imag (x,y) = y
  fun conj (x,y) = (x, ~y)
  fun plus ((u,v), (x,y) = (u+x, v+y)
end :>
sig
  type complex
  val make : real × real → complex
  val real : complex → real
  val imag : complex → real
  val conj : complex → complex
  val plus : complex × complex → complex
end
```

The type system enforces this distinction, rejecting attempts by users of the type to conflate the abstract type with its representation. With this style of abstraction values of type *complex* are not merely isomorphic to pairs of reals; they are represented identically. We dub this style of abstraction **signing**.

Both signing and sealing appear in modern functional languages as the preferred means of defining abstract types. Signing involves drawing the abstraction boundary in the types, to be enforced statically; sealing draws the boundary in the terms, to be checked at runtime, so it is no surprise that abstract types in Scheme are typically based on sealing (Matthews and Ahmed 2008). More surprisingly, abstract types in Haskell use sealing, albeit a variant in



*ICFP '08* 22-24 September 2008, Victory, BC, Canada

which the abstraction is enforced statically. While languages related to Haskell — Gofer/Hugs (Jones and Peterson 1999) and Miranda (Turner 1985) — use signing, the designers of Haskell decided in favour of sealing because it was not clear to them how to define distinct instances of a type class for an abstract type and its representation with the signing style (Hudak, Hughes, Jones, and Wadler 2007). In Standard ML both mechanisms are available, as illustrated above; as in Haskell, seals are checked statically; there is, however, no tag erasure, so abstraction violation is an error under both static and dynamic semantics.

For the language designer, then, there may be ostensible reasons to prefer one of these styles over the other. In fact, as we demonstrate in this paper, the two styles are interconvertible: there is an automatic translation between them preserving operational equivalence with respect to a standard semantics. For example, as we show in Section 2, it is possible to extend Haskell with a construct for signing by translation into the built-in constructs, avoiding the need for the user to write tags; we could equally well add such a mechanism to Scheme. In short, the language designer can safely offer either style as the means of defining abstract types with no danger of losing the benefits of the other approach except insofar as they pertain to human factors such as syntactic convenience.

Our proof that the two styles of abstract type are interconvertible is based on relational parametricity (Reynolds 1983; Wadler 1989). Pitts has developed a particularly appealing presentation of parametricity (Pitts 2000) in the presence of polymorphism and partial functions in which the usual denotational characterization of admissible relations is replaced by a purely syntactic approach. Our proof consists of an application of the central result in a minor extension to Pitts' system.

The work described in this paper treats a static variant of sealing in which programs contain a fixed number of seals, known during type-checking. This concords with the features provided by Haskell and SML, but is less general than mechanisms used in Scheme, where seals may be created dynamically. We plan to extend our result to dynamic sealing in future work.
The contributions of this paper are as follows:

1. A **characterization** of the two essential styles of abstract type, *signing* and *sealing* (Sections 1 and 3.1).
2. A **proof** that the two styles are interconvertible (Section 5) via a type-indexed function (Section 4) in a higher-order language with polymorphism and recursion, based on an extension of Pitts' PolyPCF (Section 3).
3. An **application** of the result: a robust implementation of abstract types using *signing* to Haskell (Section 2), by translation into Haskell's *sealing*-style construct, **newtype**.

## 2. Signed types in Haskell

To create an abstract type in Haskell the programmer defines a datatype in a module which does not export the data constructors. Haskell provides a special form of datatype definition, introduced with the **newtype** keyword, for creating type isomorphisms with a single, unary constructor and unlifted semantics. We can use **newtype** within a module to define the abstract type of complex numbers as follows.

```
module Complex (Complex, conj, plus,
                real, imag, make)
where
  newtype Complex = Complex (Float, Float)

  make (x,y) = Complex (x,y)
  real (Complex (x,y)) = x
  imag (Complex (x,y)) = y
  conj (Complex c) = Complex (real c, -(imag c))
  plus (Complex (u,v)) (Complex (x,y)) =
    Complex (u+x,v+y)
```

The designers of Haskell chose to provide this style of definition rather than the signing style because of concerns about how types defined with signing would interact with type classes — in particular, about potential ambiguity between type-class instances given for the representation and abstract types (Hudak et al. 2007). In this section we describe a Haskell extension written using Template Haskell (Sheard and Peyton Jones 2002) that translates abstract type definitions written in the signing style into definitions in the sealing style. With this extension the definition of the abstract type of complex numbers may be written as follows:

```
$(signed
  [d| type Complex = (Float, Float)

      make :: (Float, Float) → Complex
      make (x,y) = (x,y)

      real :: Complex → Float
      real (x,y) = x

      imag :: Complex → Float
      imag (x,y) = y

      conj :: Complex → Complex
      conj c = (real c, -(imag c))

      plus :: Complex → Complex → Complex
      plus (u,v) (x,y) = (u+x,v+y) |])
```

The Template Haskell quote operation $[d|\ldots|]$ and unquote operation $\$(\ldots)$ convert between actual code and the abstract syntax trees used to represent it. The function *signed* is the interface to our library: it maps an abstract type definition in the signed style to an equivalent definition in the sealed style. The type signatures are mandatory, since they indicate at which points the representation type should be made abstract; that is, at which points the generated code should wrap or unwrap values in a constructor. The code generated for the above definition is as follows:

**Syntax**

| | | | |
|---|---|---|---|
| $d$ | ::= | | declarations |
| | | $p = e$ **where** $d_1; \ldots d_n$ | value |
| | | **data** $T\ \alpha_1 \ldots \alpha_n = c_1 \ldots c_n$ | datatype |
| | | **newtype** $T\ \alpha_1 \ldots \alpha_n = C\ \{x :: \tau\}$ | newtype |
| | | **type** $T\ \alpha_1 \ldots \alpha_n = \tau$ | type synonym |
| | | $x :: \tau$ | signature |
| $\tau$ | ::= | | types |
| | | $\alpha$ | variable |
| | | $T\ \tau_1 \ldots \tau_n$ | constructor appl. |
| $c$ | ::= | $C\ \tau_1 \ldots \tau_n$ | constructor decl. |
| $e$ | ::= | | expressions |
| | | $x$ | variable |
| | | $C$ | constructor |
| | | $e_1\ e_2$ | application |
| | | $\lambda p_1 \ldots p_n \to e$ | abstraction |
| | | **case** $e$ **of** $p_1 \to e_1 \ldots p_n \to e_n$ | case match |
| $p$ | ::= | | patterns |
| | | $x$ | variable |
| | | $C\ p_1 \ldots p_n$ | constructor |

| | | | |
|---|---|---|---|
| $x, x_i$ | variables | $\alpha, \alpha_i$ | type variables |
| $C, C_i$ | constructors | $T, T_i$ | type constructors |

**Signs to seals**

*signed*

```
[d| type T α1 ... α1 = τ
    x1 :: τ1
    x1 = e1
    ...
    xn :: τn
    xn = en |]
=
[d| newtype T α1 ... α1 = In { out :: τ }
    (x1, ... xn) = (y1, ... yn)
    (y1, ... yn) = (H+In[τ1] x1, ... H+In[τn] xn)
      where x1 = e1; ... xn = en

    inT = in[newtype T α1 ... α1 = In { out :: τ }]
    outT = out[newtype T α1 ... α1 = In { out :: τ }]

    mapT1 = D[reify T1]
    ...
    mapTm = D[reify Tm] |]
```

*where* $\{T_1, \ldots T_m\} = tcs\,[\tau] \cup tcs\,[\tau_1] \cup \ldots \cup tcs\,[\tau_n]$

**Finding type constructors**

*In types*

$$
\begin{array}{lcll}
tcs\ r\ [\alpha] & = & \{\} & \\
tcs\ r\ [T] & = & \{\} & \text{if } T \in r \\
 & & \{T\} \cup tcs_D\ r\ [reify\ T] & \text{if } T \notin r \\
tcs\ r\ [\tau_1 \ldots \tau_n] & = & tcs\ r\ \tau_1 \cup tcs\ r\ \tau_2 &
\end{array}
$$

*In declarations*

$$
\begin{array}{lcl}
tcs_D\ r\ [\textbf{data}\ T\ \alpha_1 \ldots \alpha_n = c_1 \ldots c_n] & = & tcs_C\ r\ [c_1] \cup \ldots tcs_C\ r\ [c_n] \\
tcs_D\ r\ [\textbf{newtype}\ T\ \alpha_1 \ldots \alpha_n = C\ \{x :: \tau\}] & = & tcs\ r\ [\tau] \\
tcs_D\ r\ [\textbf{type}\ T\ \alpha_1 \ldots \alpha_n = \tau & = & tcs\ r\ [\tau]
\end{array}
$$

*In constructors*

$$tcs_C\ [C\ \tau_1 \ldots \tau_n] \quad = \quad tcs\,[\tau_1] \ldots tcs\,[\tau_n]$$

**Translation functions**

$$
\begin{array}{rcl}
H^p_T[\alpha] & = & id \\
H^p_T[T'] & = & mapT' \\
H^+_T[T] & = & inT \\
H^-_T[T] & = & outT \\
H^p_T[\tau_1\ \tau_2] & = & H^p_T[\tau_1]\ (H^p_T[\tau_2],\ H^{\bar{p}}_T[\tau_2])
\end{array}
$$

**"In" and "out" functions**

$$
\begin{array}{l}
in[\textbf{newtype}\ T\ \alpha_1 \ldots \alpha_n = C\{unC :: \tau\}] \\
\quad = \lambda f_1 \ldots f_n\ x \to C(H^+_C[\tau]\ f_1 \ldots f_n\ x) \\
\\
out[\textbf{newtype}\ T\ \alpha_1 \ldots \alpha_n = C\{unC :: \tau\}] \\
\quad = \lambda f_1 \ldots f_n\ x \to H^+_C[\tau]\ f_1 \ldots f_n\ (unC\ x)
\end{array}
$$

**Map functions**

*For declarations*

$$
\begin{array}{l}
D[\textbf{data}\ T\ \alpha_1 \ldots \alpha_n = c_1 \ldots c_n] \\
\quad = \lambda(f_1, g_1) \ldots (f_n, g_n)\ x \to \\
\qquad \textbf{case}\ x\ of\ C^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[c_1] \ldots C^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[c_n] \\
D[\textbf{newtype}\ T\ \alpha_1 \ldots \alpha_n = C\{unC :: \tau\}] \\
\quad = \lambda(f_1, g_1) \ldots (f_n, g_n)\ x \to C(T^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\tau_n]\ (unC\ x)) \\
D[\textbf{type}\ T\ \alpha_1 \ldots \alpha_n = \tau \\
\quad = \lambda(f_1, g_1) \ldots (f_n, g_n)\ y \to T^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\tau_n]\ y
\end{array}
$$

*For types*

$$
\begin{array}{rcl}
T^+(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\alpha_i] & = & f_i \\
T^-(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\alpha_i] & = & g_i \\
T^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[C] & = & mapC \\
T^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\tau_1\ \tau_2] & = & T^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\tau_1]\ (T^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\tau_2], \\
 & & \qquad T^{\bar{p}}(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\tau_2])
\end{array}
$$

*For constructors*

$$
\begin{array}{l}
C^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[C\ \tau_1 \ldots \tau_n] \\
\quad = C\ x_1 \ldots x_n \to C(T^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\tau_1]\ x_1 \ldots T^p(\overrightarrow{(f_i, g_i)}/\overrightarrow{\alpha_i})[\tau_n]\ x_n)
\end{array}
$$

**Standard map functions**

```
mapFun :: (α → β, β → α) → (γ → δ, δ → γ)
               → (α → γ)  → (β → δ)
mapFun (_, f) (g, _) h = g . h . f

mapList :: (α → β, β → α) → [α] → [β]
mapList (f, _) = map f
```

**Polarity**

$p \in \{+, -\}$ $\quad + \stackrel{\text{def}}{=} -$ $\quad - \stackrel{\text{def}}{=} +$

**Figure 1.** Translation from signing to sealing in Haskell

```
newtype Complex = In { out :: (Float, Float) }

(conj, imag, make, plus, real)
    = (conj', imag', make', plus', real')

mapFloat = id
mapComplex = λ(f1, _) (f2, _) (x1, x2) →
                ((f1 x1, f2 x2)
                   (mapFloat, mapFloat) (mapFloat, mapFloat))
map2 = (λ(f1, _) (f2, _) → λ(x1, x2) → (f1 x1, f2 x2))

inC = In . mapComplex
outC = mapComplex . out

(conj', imag', make', plus', real')
    = (mapFun (inC, outC) (inC, outC) conj,
       mapFun (inC, outC) (mapFloat, mapFloat) imag,
       mapFun
         (map2 (mapFloat, mapFloat) (mapFloat, mapFloat),
          map2 (mapFloat, mapFloat) (mapFloat, mapFloat))
         (inC, outC)
         make,
       mapFun (inC, outC)
              (mapFun (inC, outC) (inC, outC),
               mapFun (outC, inC) (outC, inC))
              plus,
       mapFun (inC, outC) (mapFloat, mapFloat) real)
    where
      imag (x, y) = y
      real (x, y) = x
      conj c = (real c, -(imag c))
      make (x, y) = (x, y)
      plus (u, v) (x, y) = ((u + x), (v + y))
```

Names that appear in faint type are fresh names generated by Template Haskell and are not accessible outside the generated code. In particular, this includes the constructor and destructor *In* and *out* that witness the type isomorphism; thus there is no way to create or examine values of type *Complex* except via the five functions in the interface.

The functions *mapFloat* and *mapComplex* are equivalent to the identity function, since the type constructors *Float* and *Complex* are both nullary. For parameterized type constructors the map functions are more interesting: a type constructor $T$ with $n$ parameters results in a function *mapT* which takes $2n$ functions, to be applied at positive and negative occurrences of each type parameter. For example, the map function for the list type, *mapList* has type $(\alpha \to \beta, \beta \to \alpha) \to [\alpha] \to [\beta]$. In this case only the first function passed to *mapList* is used, since the type parameter to the list type only occurs positively in its definition. We must distinguish between positive and negative occurrences, since at positive occurrences of the type constructor *Complex* we will insert calls to *In* to convert from the representation to the abstract type, and at negative occurrences we will insert calls to *out*. In general the library will generate a map function for each type constructor which is referred to in the definition, including those which do not appear syntactically.

For each operation $f$ in the interface, we generate a transformation function that converts from the signing-style version of the operation to the sealing-style version. For example, for the function *real*, the generated transformation function, which uses the supplied function *mapFun*

```
mapFun (_, f) (g, _) h = g . h . f
```

is the following

```
mapFun (inC, outC) (mapFloat, mapFloat)
```

with the type

```
((Float, Float) → Float) → (Complex → Float)
```

The bindings in the generated code are carefully arranged so that references to bindings from inside the abstraction resolve to the untransformed versions, while references to the bindings from outside resolve to the versions that have been transformed to use seals. In particular, the call to *real* in the definition of *conj* resolves to the function with type $(Float, Float) \to Float$, not to the version function with type $Complex \to Float$ that we expose to the user.

Figure 1 gives the general scheme for generating map and translation functions. Several operations make use of polarity; we parameterize these by a superscript $p$, which ranges over $+$ and $-$, with an operation $\overline{p}$ for switching from one to the other. The functions $tcs$, $tcs_D$ and $tcs_C$ find all type constructors used in the types, type declarations and constructor declarations; when a type constructor is encountered the corresponding definition is retrieved for examination using the *reify* operation provided by Template Haskell. (The argument $r$ is used to keep track of recursive datatype bindings in order to avoid infinite regress.) The function $D$ generates "map" functions for declarations; it makes use of auxiliary definitions $T$ and $C$ for generating sub-expressions corresponding to types and to constructors, respectively. Both $T$ and $C$ take an argument which maps type variables to pairs of functions. When a type variable is encountered, one of these functions (depending on the polarity) is returned as the generated term. At type applications (the last case of the function $T$) we generate two terms for the "argument" type, with positive and negative polarity; these are both passed as arguments to the term generated by the "function" type, to be invoked at values corresponding to positive and negative occurrences of the type argument. The function *in* takes a **newtype** declaration and generates a function that wraps values of the representation type in the constructor, after traversing subexpressions; the function *out* generates a function that strips constructors from values and traverses the result. The function $H$ generates the translation functions described earlier, which convert functions on the signed type to functions on the sealed type. Again, the polarity switch occurs at type applications, where we generate terms with both positive and negative polarity for the argument type. The *signed* function, the interface to the library, examines the declaration of the abstract type in signing style and generates an corresponding definition in the sealing style, using the operations described above.

# 3. PolyPCF with tags

In order to demonstrate the equivalence of the two styles of defining abstract types we begin by defining a programming language in which both styles can be expressed. Our language of choice is an extension of Pitts' *PolyPCF* (Pitts 2000), which was designed to investigate the operational behaviour of programs built from partial polymorphic functions. The signature style of abstract type definition can be reduced to a use case of polymorphic types, already present in PolyPCF. In order to capture the constructor style we extend PolyPCF with "tags", a sort of unary variant type with a constructor $\mathbf{in}_T$, destructor $\mathbf{out}_T$ and type constructor $T(-)$, for each $T$ of an infinite set of tag names *Tag*.

It is convenient to establish the necessary results in terms of polymorphic types and tag types, which are at once simpler and more general than the constructs for abstract type definition with which we are ultimately concerned. These latter forms, which we introduce into the syntax with the **signtype** and **sealtype** keywords, are intended to capture the essence of signature-style and sealing-style abstract type definitions; such a definition combines the operations of defining a fresh type, bringing it into scope in a particular region of the program, and —in the case of **sealtype**—associating

*Sealing*

```
sealtype α = C(number × number)
     with make : number × number → α
               = λp:number × number (in_C p)
          real : α → number
               = λp:C(number × number)(fst (out_C p))
          imag : α → number
               = λp:C(number × number)(snd (out_C p))
          conj : α → α
               = λp:C(number × number)(in_C (fst (out_C p),
                                             -(snd (out_C p))))
          plus : α → α → α
               = λp:C(number × number)
                     (λq:C(number × number)
                        (in_C(fst (out_C p) + fst (out_C q),
                              snd (out_C p) + snd (out_C q))))
       in M
```

*Signing*

```
signtype α = number × number
     with make : number × number → α
               = λp:number × number (p)
          real : α → number
               = λp:number × number(fst p)
          imag : α → number
               = λp:number × number(snd p)
          conj : α → α
               = λp:number × number((fst p, -snd p))
          plus : α → α → α
               = λp:number × number
                     (λq:number × number
                     ((fst p + fst q,
                       snd p + snd q)))
       in M
```

*Sealing, desugared*

```
(Λα(λmake:number × number → α
   (λreal:α → number
   (λimag:α → number
   (λconj:α → α
   (λplus:α → α → α
     (M)))))))
(C(number × number))
(λp:number × number (in_C p))
(λp:C(number × number)(fst (out_C p)))
(λp:C(number × number)(snd (out_C p)))
(λp:C(number × number)(in_C (fst out_C p, -snd (out_C p))))
(λp:C(number × number)(λq:C(number × number)
(in_C (fst (out_C p) + fst (out_C q),
       snd (out_C p) + snd (out_C q)))))
```

*Signing, desugared*

```
(Λα(λmake:number × number → α
   (λreal:α → number
   (λimag:α → number
   (λconj:α → α
   (λplus:α → α → α
     (M)))))))
(number × number)
(λp:number × number(p))
(λp:number × number(fst p))
(λp:number × number(snd p))
(λp:number × number((fst p, -snd p)))
(λp:number × number(λq:number × number((fst p + fst q,
                                        snd p + snd q))))
```

**Figure 2.** An abstract type of complex numbers in extended PolyPCF

a tag with the type. Investigating the properties of tag types and polymorphic types allows us to separate these operations, resulting in a simpler presentation.

In this section we illustrate by example how abstract types may be defined in PolyPCF, leaving the formal development of the language to the next section and the definition of the derived forms **sealtype** and **signtype** to Section 5.

## 3.1 Example

The four programs in Figure 2 illustrate the encoding of the two styles of type abstraction. Our examples make use of types for pairs and numbers which are not directly supported by PolyPCF, but which can be Church-encoded in the usual way. We will use pairs and numbers as though PolyPCF supported them directly.

The top-left program gives a constructor-style definition of a complex number type using **sealtype**. Since PolyPCF is explicitly typed —unlike, say, ML— we must give type signatures for each exported value. The second example gives the same definition written in plain PolyPCF without **sealtype**.

The top-right program gives a signature-style definition of a complex number type using **signtype**. The fourth example gives the same definition written in plain PolyPCF without **signtype**.

The remainder of this section is a straightforward recapitulation of (Pitts 2000). Most figures and definitions are quotes from this work, the key difference being the addition of tags. Adding tags is straightforward: no new insight or style of proof is required. The parenthesised numbers accompanying each definition and figure refer to the corresponding definition or figure in Pitts' work.

## 3.2 Syntax

Fig. 3 defines the syntax of our extension of PolyPCF. We retain the usual conventions: for example, application is left associative and we omit parentheses where possible.

PolyPCF augments the familiar polymorphic lambda calculus with fixpoint recursion and polymorphic lists, which serve the role of a ground type.

## 3.3 Typing

Figure 4 gives the typing relation $\Gamma \vdash M : \tau$ between typing environments $\Gamma$, terms $M$ and types $\tau$. As the judgements are entirely standard we refrain from commenting further.

## 3.4 Evaluation

**Definition 1. (Values).** The set $V$ of values is drawn from the set of closed terms of closed type generated by the following grammar:

$$V ::= \lambda x : \tau(M) \mid \Lambda\alpha(M) \mid \mathbf{nil}_\tau \mid M :: M \mid \mathbf{in}_T\ M$$

Figure 5 gives the evaluation relation for extended PolyPCF. The rules are mostly determined by the definition of values. Function application is call-by-name.

## 3.5 Relations

Our proof of equivalence hinges on the properties of a particular binary relation on terms, to be defined in Section 3.8. We wish to show that pairs of programs that are syntactically related in a particular way have equivalent behaviour in a precise sense. We first establish what is meant by "equivalent behaviour" (Theorem 3), then

$$
\begin{array}{lll}
\tau & ::= & \textbf{(Types)} \\
 & \alpha & \text{type variable} \\
 & \tau \to \tau & \text{function type} \\
 & \forall\alpha(\tau) & \forall - \text{type} \\
 & \tau\ \mathit{list} & \text{list type} \\
 & T(\tau) & \text{tagged type} \\
M & ::= & \textbf{(Terms)} \\
 & x & \text{variable} \\
 & \lambda x : \tau(M) & \text{function abstraction} \\
 & M\ M & \text{function application} \\
 & \Lambda\alpha(M) & \text{type generalisation} \\
 & M\ \tau & \text{type specialisation} \\
 & \mathbf{fix}(M) & \text{fixpoint recursion} \\
 & \mathbf{nil}_\tau & \text{empty list} \\
 & M :: M & \text{non-empty list} \\
 & \mathbf{case}\ M\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M & \\
 & \qquad \mid x :: x \Rightarrow M\} & \text{case expression} \\
 & \mathbf{in}_T\ M & \text{tagged constructor} \\
 & \mathbf{out}_T\ M & \text{tag destructor}
\end{array}
$$

**Notes**

(i) $\alpha$ and $x$ range over disjoint countably infinite sets *TyVar* and *Var* of type variables and variables respectively.

(ii) The constructions $\forall\alpha(-)$, $\lambda x : \tau(-)$, $\Lambda\alpha(-)$, and $\mathbf{case}\ M\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M' \mid x :: x' \Rightarrow (-)\}$ are binders. We will identify types and terms up to renaming of bound variables and bound type variables.

(iii) We write $ftv(e)$ for the finite set of free type variables of an expression $e$ (be it a type or a term) and $fv(M)$ for the finite set of free variables of a term $M$.

(iv) The result of capture-avoiding substitution of a type $\tau$ for all free occurrences of a type variable $\alpha$ in $e$ (a type or a term) will be denoted $e[\tau/\alpha]$. Similarly, $M[M'/x]$ denotes the result of capture-avoiding substitution of a term $M'$ for all free occurrences of the variable $x$ in $M$.

**Figure 3.** Syntax of the PolyPCF language extended with tags (Fig. 1)

develop a framework in which programs that are suitably syntactically related way are also semantically related, and finally show that the classes of programs with which we are concerned are syntactically related, and so behave equivalently.

**Definition 2. (Properties of relations).** (2.2) Suppose $\mathcal{E}$ is a set of 4-tuples $(\Gamma, M, M', \tau)$ satisfying

$$\Gamma \vdash M\mathcal{E}M' : \tau \Rightarrow (\Gamma \vdash M : \tau\ \&\ \Gamma \vdash M' : \tau)$$

where we write $\Gamma \vdash M\mathcal{E}M' : \tau$ instead of $(\Gamma, M, M', \tau) \in \mathcal{E}$.

(i) $\mathcal{E}$ is *compatible* if it is closed under the axioms and rules in Fig. 6. It is *substitutive* if it is closed under the rules in Fig. 7.

(ii) Compatible relations are automatically *reflexive*. A PolyPCF *precongruence* is a compatible, substitutive relation which is also transitive. A PolyPCF *congruence* is a precongruence which is also symmetric.

(iii) $\mathcal{E}$ is *adequate* if for all closed types $\tau \in Typ$ and closed terms $M, M' \in Term(\tau\ list)$

$$\emptyset \vdash M\ \mathcal{E}\ M' : \tau\ list \Rightarrow (M \Downarrow \mathbf{nil}_\tau \overset{\text{def}}{\Leftrightarrow} M' \Downarrow \mathbf{nil}_\tau)$$

**Theorem 3.** ***(PolyPCF observational congruence).*** *(2.3) There is a largest adequate, compatible and substitutive relation. It is an equivalence relation and hence is the largest adequate PolyPCF congruence relation. We call it PolyPCF observational congruence and write it as* $=_{obs}$.

$$\Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1(M) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash F : \tau_1 \to \tau_2 \qquad \Gamma \vdash A : \tau_1}{\Gamma \vdash F\ A : \tau_2}$$

$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha(M) : \forall\alpha(\tau)} \qquad \frac{\Gamma \vdash G : \forall\alpha(\tau_1)}{\Gamma \vdash G\ \tau_2 : \tau_1[\tau_2/\alpha]}$$

$$\frac{\Gamma \vdash F : \tau \to \tau}{\Gamma \vdash \mathbf{fix}(F) : \tau}$$

$$\Gamma \vdash \mathbf{nil}_\tau : \tau\ list \qquad \frac{\Gamma \vdash H : \tau \qquad \Gamma \vdash T : \tau\ list}{\Gamma \vdash H :: T : \tau\ list}$$

$$\frac{\Gamma \vdash L : \tau_1\ list \qquad \Gamma \vdash M_1 : \tau_2 \qquad \Gamma, h : \tau_1, t : \tau_1\ list \vdash M_2 : \tau_2}{\Gamma \vdash \mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\} : \tau_2}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathbf{in}_T\ M : T(\tau)} \qquad \frac{\Gamma \vdash M : T(\tau)}{\Gamma \vdash \mathbf{out}_T\ M : \tau}$$

**Notes**

(i) Typing judgements take the form $\Gamma \vdash M : \tau$ where

— the *typing environment* $\Gamma$ is a pair $A, \Delta$ with $A$ a finite subset of *TyVar* and $\Delta$ a function defined on a finite subset $dom(\Delta)$ of *Var* and mapping each $x \in dom(\Delta)$ to a type with free type variables in A;

— $M$ is a term with $ftv(M) \subseteq A$ and $fv(M) \subseteq dom(\Delta)$;;

— $\tau$ is a type with $ftv(\tau) \subseteq A$.

(ii) The notation $\Gamma, x : \tau$ indicates the typing environment obtained from the typing environment $\Gamma = A, \Delta$ by properly extending the function $\Delta$ by mapping $x \notin dom(\Delta)$ to $\tau$. Similarly, $\Gamma, \alpha$ is the typing environment obtained by properly extending $A$ with an $\alpha \notin A$.

(iii) The explicit type information included in the syntax of function abstractions and empty lists ensures that, given $\Gamma$ and $M$, there is at most one $\tau$ for which $\Gamma \vdash M : \tau$ holds.

**Figure 4.** Typing assignment relation for PolyPCF with tags (Fig. 2)

## 3.6 Frame stacks

**Definition 4. (Frame Stacks).** (3.2) *Frame stacks* provide a way to denote evaluation contexts. The grammar for PolyPCF frame stacks is

$$S ::= Id \mid S \circ F$$

where $F$ ranges over *frames*:

$$F ::= (-\ M) \mid (-\ \tau) \mid (\mathbf{case}\ -\ \mathbf{of}\{\mathbf{nil} \Rightarrow M \mid x :: x \Rightarrow M\}) \mid \mathbf{out}_T\ -$$

**Theorem 5.** ***(A structural induction principle for PolyPCF termination).*** *(3.6) For all closed types* $\tau, \tau' \in Typ$*, for all frame stacks*

$$V \Downarrow V$$

$$\frac{F \Downarrow \lambda x:\tau(M) \qquad M[A/x] \Downarrow V}{F\ A \Downarrow V} \qquad \frac{G \Downarrow \Lambda\alpha(M) \qquad M[\tau/\alpha] \Downarrow V}{G\ \tau \Downarrow V}$$

$$\frac{F\ \mathbf{fix}(F) \Downarrow V}{\mathbf{fix}(F) \Downarrow V}$$

$$\frac{L \Downarrow \mathbf{nil}_\tau \qquad M_1 \Downarrow V}{\mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\} \Downarrow V}$$

$$\frac{L \Downarrow H :: T \qquad M_2[H/h, T/t] \Downarrow V}{\mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\} \Downarrow V}$$

$$\frac{M' \Downarrow \mathbf{in}_T\ M \qquad M \Downarrow V}{\mathbf{out}_T\ M' \Downarrow V}$$

**Figure 5.** PolyPCF evaluation relation (Fig. 3)

$$\Gamma, x:\tau \vdash x\ \mathcal{E}\ x:\tau$$

$$\frac{\Gamma, x:\tau_1 \vdash M\ \mathcal{E}\ M':\tau_2}{\Gamma \vdash \lambda x:\tau_1(M)\ \mathcal{E}\ \lambda x:\tau_1(M'):\tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash F\ \mathcal{E}\ F'\tau_1 \to \tau_2 \qquad \Gamma \vdash A\ \mathcal{E}\ A':\tau_1}{\Gamma \vdash (FA)\ \mathcal{E}\ (F'A'):\tau_2}$$

$$\frac{\alpha, \Gamma \vdash M\ \mathcal{E}\ M':\tau}{\Gamma \vdash \Lambda\alpha(M)\ \mathcal{E}\ \Lambda\alpha(M'):\forall\alpha(\tau)} \qquad \frac{\Gamma \vdash G\ \mathcal{E}\ G:\forall\alpha(\tau_1)}{\Gamma \vdash (G\ \tau_2)\ \mathcal{E}\ (G'\ \tau_2):\tau_1[\tau_2/\alpha]}$$

$$\frac{\Gamma \vdash F\ \mathcal{E}\ F':\tau \to \tau}{\Gamma \vdash \mathbf{fix}(F)\ \mathcal{E}\ \mathbf{fix}(F'):\tau}$$

$$\Gamma \vdash \mathbf{nil}_\tau\ \mathcal{E}\ \mathbf{nil}_\tau : \tau\ list$$

$$\frac{\Gamma \vdash H\ \mathcal{E}\ H':\tau \qquad \Gamma \vdash T\ \mathcal{E}\ T':\tau\ list}{\Gamma \vdash (H :: T)\ \mathcal{E}\ (H' :: T'):\tau\ list}$$

$$\frac{\begin{array}{c}\Gamma \vdash L\ \mathcal{E}\ L':\tau_1\ list \qquad \Gamma \vdash M_1\ \mathcal{E}\ M_1':\tau_2 \\ \Gamma, h:\tau_1, t:\tau_1\ list \vdash M_2\ \mathcal{E}\ M_2' \vdash \tau_2\end{array}}{\begin{array}{l}\Gamma \vdash (\mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\}) \\ \quad \mathcal{E}\ (\mathbf{case}\ L'\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1' \mid h :: t \Rightarrow M_2'\}) : \tau_2\end{array}}$$

$$\frac{\Gamma \vdash M\ \mathcal{E}\ M':\tau}{\Gamma \vdash \mathbf{in}_T\ M\ \mathcal{E}\ \mathbf{in}_T\ M':T(\tau)} \qquad \frac{\Gamma \vdash M\ \mathcal{E}\ M':T(\tau)}{\Gamma \vdash \mathbf{out}_T\ M\ \mathcal{E}\ \mathbf{out}_T\ M':\tau}$$

**Figure 6.** Compatibility properties (Fig. 4)

*$S \in Stack(\tau, \tau'$ **list**), and for all closed terms $M \in Term(\tau)$, we have*

$$S\ M \Downarrow \boldsymbol{nil}_{\tau'} \quad \Leftrightarrow \quad S \top M$$

*where the relation $(-) \top (-)$ is inductively defined by the rules in Figure 9.*

## 3.7 Term and stack relations

**Definition 6. (Term- and stack-relations).** (3.8) A PolyPCF term-relation is a binary relation between (typeable) closed terms. Given closed PolyPCF types $\tau, \tau' \in Typ$, we write

$$Rel(\tau, \tau')$$

for the set of term-relations that are subsets of $Term(\tau) \times Term(\tau')$. A PolyPCF *stack-relation* is a binary relation between (typeable) frame stacks whose result types are list types. We write

$$\frac{\alpha, \Gamma \vdash M\ \mathcal{E}\ M':\tau_1}{\Gamma[\tau_2/\alpha] \vdash M[\tau_2/\alpha]\ \mathcal{E}\ M'[\tau_2/\alpha] : \tau_1[\tau_2/\alpha]}$$

$$\frac{\Gamma, x:\tau_1 \vdash M\ \mathcal{E}\ M':\tau_2 \qquad \Gamma \vdash N\ \mathcal{E}\ N':\tau_1}{\Gamma \vdash M[N/x]\ \mathcal{E}\ M'[N'/x] : \tau_2}$$

**Figure 7.** Substitutivity properties (Fig. 5)

$$\Gamma \vdash Id : \tau \circ\!\!\to \tau$$

$$\frac{\Gamma \vdash S : \tau' \circ\!\!\to \tau'' \qquad \Gamma \vdash A : \tau}{\Gamma \vdash S \circ (-A) : (\tau \to \tau') \circ\!\!\to \tau''}$$

$$\frac{\Gamma \vdash S : \tau'[\tau/\alpha] \circ\!\!\to \tau'' \qquad \alpha \text{ not free in } \Gamma}{\Gamma \vdash S \circ (-\tau) : \forall\alpha(\tau') \circ\!\!\to \tau''}$$

$$\frac{\Gamma \vdash S : \tau' \circ\!\!\to \tau'' \qquad \Gamma \vdash M_1 : \tau' \qquad \Gamma, h:\tau, t:\tau\ list \vdash M_2 : \tau'}{\Gamma \vdash S \circ (\mathbf{case}\ -\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\}) : \tau\ list \circ\!\!\to \tau''}$$

$$\frac{\Gamma \vdash S : \tau \circ\!\!\to \tau'}{\Gamma \vdash S \circ \mathbf{out}_T - : T(\tau) \circ\!\!\to \tau'}$$

**Figure 8.** Typing frame stacks (Fig. 6)

$$\frac{S = S' \circ (-A) \qquad S' \top M[A/x]}{S \top \lambda x:\tau(M)} \qquad \frac{S \circ (-A) \top F}{S \top F\ A}$$

$$\frac{S = S' \circ (-\tau) \qquad S' \top M[\tau/\alpha]}{S \top \Lambda\alpha(M)} \qquad \frac{S \circ (-\tau) \top G}{S \top G\ \tau}$$

$$\frac{S \circ (-\mathbf{fix}(F)) \top F}{S \circ \top \mathbf{fix}(F)}$$

$$\frac{S = \mathrm{Id}}{S \top \mathbf{nil}_\tau}$$

$$\frac{S = S' \circ (\mathbf{case}\ -\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\}) \qquad S' \top M_1}{S \top \mathbf{nil}_\tau}$$

$$\frac{S = S' \circ (\mathbf{case}\ -\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\}) \qquad S' \top M_2[H/h, T/t]\})}{S \top H :: T}$$

$$\frac{S \circ (\mathbf{case}\ -\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\}) \top M}{S \top \mathbf{case}\ M\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\}}$$

$$\frac{S = S' \circ (\mathbf{out}_T -) \qquad S' \top M}{S \top \mathbf{in}_T\ M} \qquad \frac{S \circ (\mathbf{out}_T -) \top M}{S \top \mathbf{out}_T\ M}$$

**Figure 9.** Structural termination relation (Fig. 7)

$$StRel(\tau, \tau')$$

for the set of stack-relations that are subsets of $Stack(\tau) \times Stack(\tau')$.

Using the $(-) \top (-)$ relation we can manufacture a stack-relation from a term-relation and vice versa, as follows:

**Definition 7. (The $(-)^\top$ operation on relations).** (3.9) Given any $\tau, \tau' \in Typ$ and $r \in Rel(\tau, \tau')$, define $r^\top \in StRel(\tau, \tau')$ by

$$(S, S') \in r^\top \stackrel{\text{def}}{\Leftrightarrow} \forall (M, M') \in r (S \top M \Leftrightarrow S' \top M');$$

and given any $s \in StRel(\tau, \tau')$ define $s^\top \in Rel(\tau, \tau')$ by

$$(M, m') \in s^\top \stackrel{\text{def}}{\Leftrightarrow} \forall (S, S') \in s (S \top M \Leftrightarrow S' \top M')$$

The $(-)^\top$ operator gives us a Galois connection with respect to inclusion, i.e.

$$r \subseteq s^\top \quad \Leftrightarrow \quad s \subseteq r^\top$$

**Definition 8. ($\top\top$-Closed term-relations).** (3.10) A term-relation $r$ is $\top\top$-*closed* if $r = r^{\top\top}$, or equivalently if $r^{\top\top} \subseteq r$, or equivalently if $r = s^\top$ for some stack-relation $s$, or equivalently if $r = (r')^{\top\top}$ for some term-relation $r'$.

## 3.8 Action of type constructors on term relations

We are now ready to define the central logical relation $\Delta$. Each type constructor in PolyPCF has an associated "action" on term relations. The combination of these gives the definition of a logical relation, parameterized by a tuple of term-relations.

**Definition 9. (Action of $\to$ on term-relations).** (4.1) Given $r_1 \in Rel(\tau_1, \tau_1')$ and $r_2 \in Rel(\tau_2, \tau_2')$, we define $r_1 \to r_2 \in Rel(\tau_1 \to \tau_2, \tau_1' \to \tau_2')$ by:

$$(F, F') \in r_1 \to r_2 \stackrel{\text{def}}{\Leftrightarrow} \forall (A, A') \in r_1 ((F\ A, F'\ A') \in r_2).$$

**Definition 10. (Action of $\forall$ on term-relations).** (4.2) Let $\tau_1$ and $\tau_1'$ be PolyPCF types with at most a single free type variable, $\alpha$ say. Suppose $R$ is a function mapping term-relations $R \in Rel(\tau_2, \tau_2')$ (any $\tau_2, \tau_2' \in Typ$) to term-relations $R(r) \in Rel(\tau_1[\tau_2/\alpha], \tau_1'[\tau_2'/\alpha])$. Then we can form a term-relation $\forall r(R(r)) \in Rel(\forall\alpha(\tau_1), \forall\alpha(\tau_1'))$ as follows:

$$(G, G') \in \forall r(R(r)) \stackrel{\text{def}}{\Leftrightarrow} \forall \tau_2, \tau_2' \in Typ (\forall r \in Rel(\tau_2, \tau_2') \\ ((G\ \tau_2, G'\ \tau_2') \in R(r))).$$

**Definition 11. (Action of ( list$-$) on term-relations).** (4.3) Given $\tau, \tau' \in Typ$, $r_1 \in Rel(\tau, \tau')$ and $r_2 \in Rel(\tau\ \mathbf{list}, \tau'\ \mathbf{list})$, define $1 + (r_1 \times r_2) \in Rel(\tau\ \mathbf{list}, \tau'\ \mathbf{list})$ by:

$$1 + (r_1 \times r_2) \stackrel{\text{def}}{=} \\ \{(\mathbf{nil}_\tau, \mathbf{nil}_{\tau'})\} \cup \{(H :: T, H' :: T') \mid (H, H') \in r_1\ \&\ (T, T') \in r_2\}$$

Note that the subset relation makes $Rel(\tau\ \mathbf{list}, \tau'\ \mathbf{list})$ into a complete lattice and that, for each $r_1$ the function $r_2 \mapsto (1 + (r_1 \times r_2))^{\top\top}$ is monotone. Therefore we can form its greatest (post-)fixed point:

$$(r_1)\ \mathbf{list} \stackrel{\text{def}}{=} \nu r_2 (1 + (r_1 \times r_2))^{\top\top}.$$

Thus $(r_1)$ **list** is the unique term-relation satisfying

$$(r_1)\ \mathbf{list} = (1 + (r_1 \times (r_1)\ \mathbf{list}))^{\top\top}$$
$$\forall r_2 (r_2 \subseteq (1 + (r_1 \times r_2))^{\top\top} \Rightarrow r_2 \subseteq (r_1)\ \mathbf{list}$$

**Definition 12. (Action of $T(-)$ on term relations).** Given $r \in Rel(\tau, \tau')$ define $T(r) \in Rel(T(\tau), T(\tau'))$ by

$$T(r) \stackrel{\text{def}}{=} \{(M, M') | (\mathbf{out}_T\ M, \mathbf{out}_{M'}) \in r\}$$

**Definition 13. (The logical relation $\Delta$).** For each PolyPCF type *tau* and each list $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ of distinct type variables containing the free type variables of $\tau$, we define a function from tuples of term-relations to term-relations

$$r_1 \in Rel(\tau_1, \tau_1'), \ldots, r_n \in Rel(\tau_n, \tau_n') \mapsto \Delta_\tau(\vec{r}/\vec{\alpha}) \in Rel(\tau[\vec{\tau}/\vec{\alpha}], \tau[\vec{\tau'}/\vec{\alpha'}]).$$

where $\Delta$ is defined as in Figure 10.

**Definition 14. (Logical relation on open terms).** (4.5) Suppose $\Gamma \vdash M : \tau$ and $\Gamma \vdash M' : \tau$ hold, with $\Gamma = \alpha_1, \ldots, \alpha_m, x_1 : \tau_1, \ldots, x_n : \tau_n$ say. Write

$$\Gamma \vdash M \Delta M' : \tau \tag{1}$$

to mean: given any $\sigma_i, \sigma_i' \in Typ$ and $r_i \in Rel(\sigma_i, \sigma_i')$ (for $i = 1..m$) with each $r_i$ $\top\top$-closed, then for any $(N_j, N_j') \in \Delta_{r_j}(\vec{r}/\vec{\alpha})$ (for $i = 1..n$) it is the case that

$$(M[\vec{\sigma}/\vec{\alpha}, \vec{N}\vec{x}], M'[\vec{\sigma'}/\vec{\alpha}, \vec{N'}\vec{x}]) \in \Delta_\tau(\vec{r}/\vec{\alpha})$$

$$\begin{array}{rcl}
\Delta_{\alpha_i}(\vec{r}/\vec{\alpha}) & \stackrel{\text{def}}{=} & r_i \\
\Delta_{\tau \to \tau'}(\vec{r}/\vec{\alpha}) & \stackrel{\text{def}}{=} & \Delta_\tau(\vec{r}/\vec{\alpha}) \to \Delta_{\tau'}(\vec{r}/\vec{\alpha}) \\
\Delta_{\forall\alpha(\tau)'}(\vec{r}/\vec{\alpha}) & \stackrel{\text{def}}{=} & \forall r (\Delta_\tau(r^{\top\top}/\alpha, \vec{r}/\vec{\alpha})) \\
\Delta_{\tau\ list}(\vec{r}/\vec{\alpha}) & \stackrel{\text{def}}{=} & (\Delta_\tau(\vec{r}/\vec{\alpha}))\ list \\
\Delta_{T(\tau)}(\vec{r}/\vec{\alpha}) & \stackrel{\text{def}}{=} & T(\Delta_\tau(\vec{r}/\vec{\alpha}))
\end{array}$$

**Figure 10.** Definition of the logical relation $\Delta$ (Fig. 8)

## 3.9 Fundamental Property

**Proposition 15.** ***('Fundamental Property' of the logical relation).*** *(4.6) The relation (1) between open PolyPCF terms is compatible and substitutive, in the sense of Definition 2.*

The proof of Proposition 15 depends on the following Lemma:

**Lemma 16.** (4.11) For each open type $\tau$, with free type variables in $\vec{\alpha}$ say, if the rem-relations $\vec{r}$ are $\top\top$-closed, then so is the term-relation $\Delta_r(\vec{r}/\vec{\alpha})$ defined in Figure 10. In particular for each closed type $\tau$, $\Delta_\tau() \in Rel(\tau, \tau)$ is $\top\top$-closed.

Pitts' proof of this lemma proceeds by induction on the structure of types, showing that the action of each type constructor takes $\top\top$-closed relations to $\top\top$-closed relations. We have added a family of type constructors for tagged types $T(-)$ and therefore need the following lemma to extend this theorem to our augmented system.

**Lemma 17** ($T(-)$ preserves $\top\top$-closure)**.** Suppose $r \in Rel(\tau, \tau')$.

(i) Suppose given values $\mathbf{in}_T\ M$ and $\mathbf{in}_T\ M'$ of types $T(\tau)$ and $T(\tau')$ respectively, satisfying $(M, M') \in r$.
If $r$ is $\top\top$-closed then $(\mathbf{in}_T\ M, \mathbf{in}_T\ M') \in T(r)$.

(ii) If $(S, S') \in r^\top$ then $(S \circ \mathbf{out}_T -, S' \circ \mathbf{out}_T -) \in T(r)^\top$.

(iii) If $r$ is $\top\top$-closed then so is $T(r)$.

*Proof.*

(i) The statement in (i) follows from the equivalence

$$\mathbf{out}_T\ \mathbf{in}_T\ M \Downarrow V \iff M \Downarrow V$$

(ii) Suppose $(S, S') \in r^\top$. For any $(N, N') \in T(r)$ we have

$$\begin{array}{rcl}
S \circ \mathbf{out}_T - \top N & \Leftrightarrow & S \top \mathbf{out}_T\ N \\
& & \quad \text{(by definition of } (-) \top (-)) \\
& \Leftrightarrow & S' \top \mathbf{out}_T\ N' \\
& & \quad \text{(since } (\mathbf{out}_T\ N, \mathbf{out}_T\ N') \in r \\
& & \qquad \text{and } (S, S') \in r^\top) \\
& \Leftrightarrow & S' \circ \mathbf{out}_T - \top N' \\
& & \quad \text{( by definition of } (-) \top (-))
\end{array}$$

Thus $(S \circ \mathbf{out}_T -, S' \circ \mathbf{out}_T -) \in T(r)^\top$.

(iii) Suppose $(S, S') \in r^\top$, $(N, N') \in r$, $(M, M') \in T(r)^{\top\top}$. We must show that $(M, M') \in T(r)$.
By (ii) we have $(S \circ \mathbf{out}_T -, S' \circ \mathbf{out}_T -) \in T(r)^\top$, and hence

$$S \circ \mathbf{out}_T - \top M \Leftrightarrow S' \circ \mathbf{out}_T - \top M'$$

Therefore

$$S \top \mathbf{out}_T\ M \Leftrightarrow S' \top \mathbf{out}_T\ M'$$

Thus, by the definition of $(-)^\top$, $(\mathbf{out}_T\, M, \mathbf{out}_T\, M') \in (r^\top)^\top = r$, and so $(M, M') \in T(r)$ by the definition of $T(-)$. Thus $T(r)$ is $\top\top$-closed.

□

**Proposition 18.** *(4.15) The logical relation (1) coincides with PolyPCF observational congruence:*

$$\Gamma \vdash M \overset{obs}{=} M' : \tau \Leftrightarrow \Gamma \vdash M \Delta M' : \tau \tag{2}$$

Having established the necessary preliminaries to the equivalence proof, we now turn to the development of the conversion between the two styles of abstract type.

# 4. Tagging and untagging

## 4.1 An example

The conversion between signed and sealed definitions of abstract types is performed by a pair of type-indexed functions which insert and remove tags as necessary. For example, to convert the function *plus* in the signed implementation of the complex type given in Section 3.1 into a function that can be used in the sealed implementation we generate a function of type

$$(\alpha \to \alpha \to \alpha) \to (C(\alpha) \to C(\alpha) \to C(\alpha))$$

where $\alpha$ is the type variable denoting the abstract type of complex numbers and $C$ is the tag used in the sealed representation. The conversion function we generate based on the type of *plus* in the sealed representation is

$$\begin{array}{l}\lambda h : (\alpha \to \alpha \to \alpha) \\ \quad (\lambda x : C(\alpha) \\ \qquad ((\lambda j : \alpha \to \alpha \\ \qquad\quad (\lambda y : C(\alpha)\,((\lambda z : \alpha\,(\mathbf{in}_C\ z)) \\ \qquad\qquad\qquad (j\ ((\lambda z : C(\alpha)\,(\mathbf{out}_C\ z))\ y))))) \\ \qquad (h\ ((\lambda z : C(\alpha)\,(\mathbf{out}_C\ z))\ x))))\end{array}$$

or, after performing some "administrative reductions" (Plotkin 1975),

$$\begin{array}{l}\lambda h : (\alpha \to \alpha \to \alpha) \\ \quad (\lambda x : C(\alpha)\,(\lambda y : C(\alpha)\,(\mathbf{in}_C\ (h\ (\mathbf{out}_C\ x)\ (\mathbf{out}_C\ y)))))\end{array}$$

Conversely, to move from the sealed to the signed representation we generate a function of type

$$(C(\alpha) \to C(\alpha) \to C(\alpha)) \to (\alpha \to \alpha \to \alpha)$$

The generated term, after administrative reductions, is

$$\begin{array}{l}\lambda h : (C(\alpha) \to C(\alpha) \to C(\alpha)) \\ \quad (\lambda x : \alpha\,(\lambda y : \alpha\,(\mathbf{out}_C\ (h\ (\mathbf{in}_C\ x)\ (\mathbf{in}_C\ y)))))\end{array}$$

## 4.2 Definition

We now give functions that translate between tagged and untagged representations of abstract types.

For each a type $\tau$ with a free variable $\alpha$ and no occurrences of the tag $T$ we will define functions $C^+_{T,\alpha}[\tau]$ and $C^-_{T,\alpha}[\tau]$ with types

$$\begin{array}{rcl} C^+_{T,\alpha}[\tau] & : & \tau \to \tau[T(\alpha)/\alpha] \\ C^-_{T,\alpha}[\tau] & : & \tau[T(\alpha)/\alpha] \to \tau \end{array}$$

and define operations $-^+_{T,\alpha}$ and $-^-_{T,\alpha}$ on types:

$$\begin{array}{rcl} \tau^+_{T,\alpha} & = & \tau \\ \tau^-_{T,\alpha} & = & \tau[T(\alpha)/\alpha] \end{array}$$

As the types suggest, these operations may be obtained for a type $\forall\alpha_(\tau)$ in the following manner: Let $\mathrm{map}_\tau$ be the map of the bifunctor corresponding to $\tau$, with type

$$\forall\beta(\forall\gamma(\beta_{\to}\gamma, \gamma \to \beta) \to \tau\,\beta \to \tau\,\gamma$$

(In general, if $\tau$ has $n$ free type variables, $\mathrm{map}_\tau$ will take $2n$ functions, since each may occur positively or negatively). Let *in* and *out* be defined as follows.

$$\begin{array}{rcl} in & \overset{\text{def}}{=} & \lambda x : \alpha(\mathbf{in}_T\ x) \\ out & \overset{\text{def}}{=} & \lambda x : T(\alpha)(\mathbf{out}_T\ x) \end{array}$$

Then

$$\begin{array}{rcl} C^+_{T,\alpha}[\tau] & \overset{\text{def}}{=} & \lambda x : \tau^+_{T,\alpha}(\mathrm{map}_{\tau'}\ \alpha\ (T(\alpha))\ in\ out) \\ C^-_{T,\alpha}[\tau] & \overset{\text{def}}{=} & \lambda x : \tau^-_{T,\alpha}(\mathrm{map}_{\tau'}\ (T(\alpha))\ \alpha\ out\ in) \end{array}$$

We now give a precise definition of $C^+_{T,\alpha}[\tau]$ and $C^-_{T,\alpha}[\tau]$ by cases on $\tau$. Let $p$ range over $\{+,-\}$ and write $\bar{p}$ for the operation that flips the polarity, so that

$$\overline{+} \overset{\text{def}}{=} - \qquad\qquad \overline{-} \overset{\text{def}}{=} +$$

To avoid excessive clutter we leave the subscript $_{T,\alpha}$ implicit. (The subscript remains constant throughout the transformation.)

$$\begin{array}{rcl} C^p[\forall\beta(\tau)] & = & \lambda x : \forall\beta(\tau)^p(\Lambda\beta(C^p[\tau]\,(x\,\beta))) \\ C^p[\tau_1 \to \tau_2] & = & \lambda h : (\tau_1 \to \tau_2)^p(\lambda x : \tau_1^{\bar{p}}(C^p[\tau_2]\,(h\,(C^{\bar{p}}[\tau_1]\ x)))) \\ C^p[\tau\ \mathbf{list}] & = & \mathrm{mapList}\ (\tau^p)\ (\tau^{\bar{p}})\ (C^p[\tau]) \\ C^p[T'(\tau)] & = & \lambda x : T'(\tau)^p(\mathbf{in}_T\ (C^p[\tau](\mathbf{out}_T\ x)) \\ C^p[\beta] & = & \lambda x : \beta(x) \\ C^+[\alpha] & = & \lambda x : \alpha(\mathbf{in}_T\ x) \\ C^-[\alpha] & = & \lambda x : T(\alpha)(\mathbf{out}_T\ x) \end{array}$$

where $\mathrm{mapList} : \forall\alpha(\forall\beta((\alpha \to \beta) \to (\alpha\ \mathbf{list} \to \beta\ \mathbf{list})))$ is defined in the usual way.

**Proposition 19.** *The functions $C^-_{T,\alpha}[\tau]$ and $C^+_{T,\alpha}[\tau]$ are mutual inverses. That is,*

$$C^-_{T,\alpha}[\tau] \circ C^+_{T,\alpha}[\tau] \overset{obs}{=} \lambda x : \tau(x) \tag{3}$$

$$C^+_{T,\alpha}[\tau] \circ C^-_{T,\alpha}[\tau] \overset{obs}{=} \lambda x : \tau[T(\alpha)/\alpha](x) \tag{4}$$

*Proof.* Either directly using the standard $\beta$- and $\eta$- equalities that follow from the evaluation rules, or indirectly via map fusion. □

# 5. Signing and sealing

In Section 3.1 we introduced the constructs **sealtype** and **signtype** for creating abstract types. We now give formal definitions of each construct, both directly and via desugaring into the core language.

## Syntax

$$\begin{array}{l} M ::= \ldots \\ \quad \mid \mathbf{sealtype}\ \alpha = T(\tau)\ \mathbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \mathbf{in}\ M \\ \quad \mid \mathbf{signtype}\ \ \alpha = \ \ \tau \quad \mathbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \mathbf{in}\ M \end{array}$$

As we noted in the introduction, the distinction between the two constructs is somewhat obscured in PolyPCF by the need to supply type signatures for each exported component in both the **signtype** and **sealtype** variants. In a language with type inference the signatures can be omitted in **sealtype** definitions.

### Typing

$$\frac{\Gamma \vdash M_i : \tau_i[T(\tau)/\alpha] \quad (\forall i.1 \le i \le n) \qquad \Gamma, \alpha, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau'' \qquad \alpha \notin \mathrm{fv}(\tau'')}{\Gamma \vdash \textbf{sealtype}\ \alpha = T(\tau)\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\ \textbf{in}\ M : \tau''}$$

$$\frac{\Gamma \vdash M_i : \tau_i[\tau/\alpha] \quad (\forall i.1 \le i \le n) \qquad \Gamma, \alpha, x_1 : \tau_1, \ldots, x_n : \tau'_n \vdash M : \tau'' \qquad \alpha \notin \mathrm{fv}(\tau'')}{\Gamma \vdash \textbf{signtype}\ \alpha = \tau\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\ \textbf{in}\ M : \tau''}$$

### Evaluation

$$\frac{M[T(\tau)/\alpha, M_1/x_1, \ldots M_n/x_n] \Downarrow V}{\textbf{sealtype}\ \alpha = T(\tau)\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\ \textbf{in}\ M \Downarrow V}$$

$$\frac{M[\tau/\alpha, M_1/x_1, \ldots M_n/x_n] \Downarrow V}{\textbf{signtype}\ \alpha = \tau\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\ \textbf{in}\ M \Downarrow V}$$

### Desugaring

The **sealtype** and **signtype** are derived forms; there is a straightforward translation into core PolyPCF terms as shown by the following lemma.

**Lemma 20** (Desugaring)**.** Let $I$, $J$, $K$ and $L$ be defined as follows for some non-negative integer $n$, terms $M, M_1, \ldots M_n$ and types $\tau, \tau_1, \ldots \tau_n$.

$$\begin{array}{lcl} I & \stackrel{\text{def}}{=} & \textbf{sealtype}\ \alpha = T(\tau)\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\ \textbf{in}\ M \\ J & \stackrel{\text{def}}{=} & \textbf{signtype}\ \alpha = \tau\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\ \textbf{in}\ M \\ K & \stackrel{\text{def}}{=} & (\Lambda\alpha(\lambda x_1 : \tau_1(\ldots \lambda x_n : \tau_n(M))\ldots))\,(T(\tau))\ M_1\ \ldots\ M_n \\ L & \stackrel{\text{def}}{=} & (\Lambda\alpha(\lambda x_1 : \tau_1(\ldots \lambda x_n : \tau_n(M))\ldots))\,(\tau)\ M_1\ \ldots\ M_n \end{array}$$

Then the following hold:

$$I \Downarrow V \iff K \Downarrow V \tag{5}$$
$$J \Downarrow V \iff L \Downarrow V \tag{6}$$
$$\Gamma \vdash I : \tau'' \iff \Gamma \vdash K : \tau'' \tag{7}$$
$$\Gamma \vdash J : \tau'' \iff \Gamma \vdash L : \tau'' \tag{8}$$

where $\alpha \notin fv(\tau'')$.

*Proof.* The statements 5 and 6 follow directly from the definition of the evaluation relation $\Downarrow$ above and in Figure 5.

The statements 7 and 8 follow directly from the definition of the typing relation $|-\ :$ above and in Figure 4. □

### 5.1 Equivalence

**Proposition 21.** *Abstract type definitions with sealtype may be converted to observationally-equivalent signtype definitions and conversely as shown below.*

$$\textbf{\textit{sealtype}}\ \alpha = T(\tau)\ \textbf{\textit{with}}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \textbf{\textit{in}}\ M \overset{obs}{=} \textbf{\textit{signtype}}\ \alpha = \tau\ \textbf{\textit{with}}\ x_1 : \tau_1 = C^-_{T,\alpha}\,[\tau_1]\,M_1 \ldots x_n : \tau_n = C^-_{T,\alpha}\,[\tau_n]\,M_n\ \textbf{\textit{in}}\ M$$

$$\textbf{\textit{signtype}}\ \alpha = \tau\ \textbf{\textit{with}}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \textbf{\textit{in}}\ M \overset{obs}{=} \textbf{\textit{sealtype}}\ \alpha = T(\tau)\ \textbf{\textit{with}}\ x_1 : \tau_1 = C^+_{T,\alpha}\,[\tau_1]\,M_1 \ldots x_n : \tau_n = C^+_{T,\alpha}\,[\tau_n]\,M_n\ \textbf{\textit{in}}\ M$$

*Proof.* Let

$$\mathbb{R}_T[\tau] = \{\langle M,\ T(M)\rangle \mid \quad \vdash M : \tau\}^{\top\top}$$

and

$$\alpha \vdash M_i : \tau_i$$

for each $1 \le i \le n$.

Comparing the definitions of $\Delta$, $C^+$ and $C^-$ reveals that

$$\langle M_i,\ C^+_{T,\alpha}[\tau_i]\,M_i\rangle \in \Delta_{\tau_i}(\mathbb{R}_T[\tau]/\alpha) \tag{9}$$
$$\langle C^-_{T,\alpha}[\tau_i]\,M_i,\ M_i\rangle \in \Delta_{\tau_i}(\mathbb{R}_T[\tau]/\alpha) \tag{10}$$

By Theorem 15 ("Fundamental Property") $\Delta$ is compatible, and hence reflexive. We can instantiate the relation (1) to give

$$\alpha, x_1 : \tau_1, \ldots x_n : \tau_n \vdash M \Delta M : \tau''$$

From (9) and (10) it follows that

$$\langle M[\tau/\alpha, \overrightarrow{M_i/x_i}],\ M[T(\tau)/\alpha, \overrightarrow{C^+_{T,\alpha}[\tau_i]\,M_i/x_i}]\rangle \in \Delta_{\tau''}(\mathbb{R}_T[\tau]/\alpha)$$
$$\langle M[\tau/\alpha, \overrightarrow{C^-_{T,\alpha}[\tau_i]\,M_i/x_i}],\ M[T(\tau)/\alpha, \overrightarrow{M_i/x_i}]\rangle \in \Delta_{\tau''}(\mathbb{R}_T[\tau]/\alpha)$$

and so (since $\alpha \notin \mathrm{fv}(\tau'')$),

$$\langle M[\tau/\alpha, \overrightarrow{M_i/x_i}],\ M[T(\tau)/\alpha, \overrightarrow{C^+_{T,\alpha}[\tau_i]\,M_i/x_i}]\rangle \in \Delta_{\tau''}()$$
$$\langle M[\tau/\alpha, \overrightarrow{C^-_{T,\alpha}[\tau_i]\,M_i/x_i}],\ M[T(\tau)/\alpha, \overrightarrow{M_i/x_i}]\rangle \in \Delta_{\tau''}()$$

and hence (by Theorem 18),

$$M[\tau/\alpha, \overrightarrow{M_i/x_i}] \overset{\text{obs}}{=} M[T(\tau)/\alpha, \overrightarrow{C^+_{T,\alpha}[\tau_1]\,M_i/x_i}]$$
$$M[T(\tau)/\alpha, \overrightarrow{M_i/x_i}] \overset{\text{obs}}{=} M[\tau/\alpha, \overrightarrow{C^-_{T,\alpha}[\tau_i]\,M_i/x_i}]$$

and hence (by the evaluation rules for **sealtype** and **signtype**),

$$\textbf{sealtype}\ \alpha = T(\tau)\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \textbf{in}\ M \overset{\text{obs}}{=} \textbf{signtype}\ \alpha = \tau\ \textbf{with}\ x_1 : \tau_1 = C^-_{T,\alpha}\,[\tau_1]\,M_1 \ldots x_n : \tau_n = C^-_{T,\alpha}\,[\tau_n]\,M_n\ \textbf{in}\ M$$

$$\textbf{signtype}\ \alpha = \tau\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \textbf{in}\ M \overset{\text{obs}}{=} \textbf{sealtype}\ \alpha = T(\tau)\ \textbf{with}\ x_1 : \tau_1 = C^+_{T,\alpha}\,[\tau_1]\,M_1 \ldots x_n : \tau_n = C^+_{T,\alpha}\,[\tau_n]\,M_n\ \textbf{in}\ M$$

□

# 6. Related work

The sealing style for creating abstract types dates back to Morris (James H. Morris 1973), who informally outlines the connection with the signing style. More recently, several authors have given formal translations between static and dynamic schemes for preserving types. It is not surprising that the translations are similar in each case, although their motivations (and consequently the precise properties that they investigate) differ considerably.

Sumii and Pierce (Pierce and Sumii 2000) have examined the relationship between type abstraction and cryptography, developing a theory of relational parametricity for their cryptographic $\lambda$-calculus. They give an encoding of type abstraction into encryption (but not vice versa); their encryption primitives, which give a variant of dynamic sealing, are significantly more expressive than our tags, so the inverse translation is not straightforward.

Matthews and Findler (Matthews and Findler 2007) investigate the semantics of programs written in multiple languages — partly in Scheme and partly in ML. They investigate two ways to encode foreign values: as opaque "lumps" that must be explicitly passed to the language which created them each time they are used, or as values that have been wrapped using a type-directed strategy, so that they can be used directly. The wrappings, which dynamically check that the values they wrap have the appropriate shapes, are an instance of higher-order contracts (Findler and Felleisen 2002); the type-directed scheme for inserting guards is analogous to our translation from signing-style to sealing-style abstract types.

Matthews and Ahmed (Matthews and Ahmed 2008) extend Matthews and Findler's system with polymorphism, and prove a parametricity property. In order to such prevent violations of parametricity as can arise from dynamic inspection of values by the Scheme portion of a program, they dynamically seal values of abstract type before passing them to Scheme.

## Acknowledgments

We thank Sam Lindley and Ezra Cooper for comments on a draft of this paper.

## References

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: http://doi.acm.org/10.1145/581478.581484.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *HOPL III*, New York, NY, USA, 2007. ACM.

Jr. James H. Morris. Types are not sets. In *POPL '73*, pages 120–124, New York, NY, USA, 1973. ACM.

Mark P. Jones and John Peterson. *The Hugs 98 User Manual*, 1999. http://cvs.haskell.org/Hugs/downloads/hugs.pdf.

Jacob Matthews and Amal Ahmed. Parametric Polymorphism Through Run-Time Sealing, or, Theorems for Low, Low Prices! In *ESOP*, March 2008.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *SIGPLAN Not.*, 42(1):3–10, 2007.

Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism, July 2000. Some parts superseded by (Sumii and Pierce 2003).

Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical. Structures in Comp. Sci.*, 10(3):321–359, 2000.

Gordon Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

John C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing '83*, pages 513–523, 1983.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4):521–554, 2003. Extended abstract appeared in *14th IEEE Computer Security Foundations Workshop*, pp. 256–269, 2001.

D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

Philip Wadler. Theorems for free! In *FPCA '89*, pages 347–359, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0.