Intensional Functions

ZACHARY PALMER, Swarthmore College, USA
NATHANIEL WESLEY FILARDO, Microsoft, Canada
KE WU, Johns Hopkins University, USA

Functions in functional languages have a single elimination form — application — and cannot be compared, hashed, or subjected to other non-application operations. These operations can be approximated via defunctionalization: functions are replaced with first-order data and calls are replaced with invocations of a dispatch function. Operations such as comparison may then be implemented for these first-order data to approximate e.g. deduplication of continuations in algorithms such as unbounded searches. Unfortunately, this encoding is tedious, imposes a maintenance burden, and obfuscates the affected code.

We introduce an alternative in *intensional functions*, a language feature which supports the definition of non-application operations in terms of a function's definition site and closure-captured values. First-order data operations may be defined on intensional functions without burdensome code transformation. We give an operational semantics and type system and prove their formal properties. We further define *intensional monads*, whose Kleisli arrows are intensional functions, enabling monadic values to be similarly subjected to additional operations.

CCS Concepts: • **Software and its engineering** → Control structures; **Functional languages**; *Coroutines*; **Constraints**; **Procedures, functions and subroutines**; *Data types and structures.*

Additional Key Words and Phrases: function, intensional, closure, continuation, defunctionalization

ACM Reference Format:
Zachary Palmer, Nathaniel Wesley Filardo, and Ke Wu. 2024. Intensional Functions. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 274 (October 2024), 26 pages. https://doi.org/10.1145/3689714

1 Introduction

Defunctionalization as proposed by Reynolds [Reynolds 1972] is the process of transforming a program to replace first-class functions with non-function symbol values. The transformation also provides a dispatch function which recovers the behavior of a function given its symbol. Higher-order function calls are replaced with invocations of this dispatch function. While defunctionalization has a variety of uses in program analysis and compiler design, we focus here on its application as a programmer-managed design pattern in functional software engineering [Danvy and Nielsen 2001; Koppel 2019]. Programmers may defunctionalize surface-level code so that operations unavailable to functions, such as equality or serialization, can be defined on first-order function symbols. This is of particular relevance to algorithms representing work as continuations: equality might be used to deduplicate continuation symbols while serialization might be used to persist them for later resumption or render them for transmission across a distributed system.

While defunctionalization is a powerful tool, its manual application to surface-level code is unfortunately tedious, error-prone, and quite obfuscating. Projects such as CloudHaskell [Epstein

Authors' Contact Information: Zachary Palmer, zachary.palmer@swarthmore.edu, Swarthmore College, Swarthmore, Pennsylvania, USA; Nathaniel Wesley Filardo, nfilardo@microsoft.com, Microsoft, Montréal, Quebec, Canada; Ke Wu, kwu48@jhu.edu, Johns Hopkins University, Baltimore, Maryland, USA.

ACM 2475-1421/2024/10-ART274
https://doi.org/10.1145/3689714

et al. 2011] and Scala's Spores [Miller et al. 2014] have addressed these weaknesses specifically for the case of serialization. Both projects allow approrpiately-annotated functions in their respective languages to be serialized and transmitted to other processes with minimal syntactic overhead. As these projects focus on serialization, however, the functions' serialized closures are not accessible to programmers with any meaningful type information.

This paper introduces *intensional functions*: functions with language-level support for general user-defined operations over dynamic closures with programmer-visible types. These functions are intensional in that they can be inspected at runtime in terms of their construction: intensional functions are equipped with an eliminator which yields the program point at which the function was defined and another eliminator which produces the values it has captured in closure. In contrast, traditional functions are *extensional*: they cannot be examined at runtime and can only be called.

Additionally, intensional functions carry programmer-specified proofs (via type constraints) about their closure-captued values. This information permits a programmer to define operations on intensional functions in terms of these proofs: equality on intensional functions, for instance, may be defined in terms of equality on the contents of their closures (with some care as described in Section 2.3). Other operations such as sorting and hashing may be defined similarly. Proofs captured by an intensional function are specified by the programmer, so this model adapts to the needs of each program's problem domain. Unlike existing approaches, there are no restrictions on intensional functions' closures other than user-specified type constraints.

Section 2 gives a description of intensional functions by example. The code in that section is written using the syntax of `IntensionalFunctions`, a Haskell language extension we have implemented for version 9.2 of the Glasgow Haskell Compiler (GHC). Throughout this paper, we refer to the Haskell language with this extension enabled as "Haskell+`ItsFn`". We find that a deductive closure algorithm of the Plume program analysis [Fachinetti et al. 2020] written using intensional functions in Haskell+`ItsFn` requires 25% fewer lines of code and is subjectively more readable than the same algorithm written using defunctionalization in Haskell. We note that our implementation is a proof of concept: it illustrates the coherence and ergonomic convenience of intensional functions but does not integrate them fully into the language runtime, resulting in significant slowdown (~3x in our experience). We believe this poor performance to be a consequence of engineering rather than theory and discuss Haskell+`ItsFn` in greater detail in Section 6.

Because intensional functions are both general and language-supported, we are also able to explore use cases which are infeasible or impossible with existing approaches. Section 3 briefly examines *intensional monads*, a reconstruction of the functor hierarchy using intensional functions. Just as extensional monad expressions represent computations as terms, intensional monad expressions represent computations as *terms subject to constraints* (such as Haskell's `Ord`).

While our motivating examples are written in Haskell+`ItsFn`, the underlying principles of intentional functions are not language-specific. Sections 4 and 5 distill these principles to a core language, λ_{ITS}, and use it to prove the equivalence of intensional functions that have the same program point and environment. We give a type system for λ_{ITS} based upon established techniques and prove it sound in this paper's supplemental material.

In summary, the primary contributions of this paper are

- a presentation and qualitative analysis of the expressive power of intensional functions (Section 2);
- the development of intensional monads, a reconstruction of the functor hierarchy using intensional functions (Section 3);
- a formal treatment of intensional functions with a correctness proof for conservative in-language function equivalence (Sections 4 and 5); and

```haskell
import Data.Map (Map)
import qualified Data.Map as Map

data Cache a b = Cache (a -> b) (Map a b)

makeCache :: (a -> b) -> Cache a b
makeCache fn = Cache fn Map.empty

apCache :: (Ord a) => Cache a b -> a -> (b, Cache a b)
apCache cache@(Cache fn m) arg =
  case Map.lookup arg m of
    Just answer -> (answer, cache)
    Nothing ->
      let answer = fn arg in
      (answer, Cache fn $ Map.insert arg answer m)
```

Fig. 1. Simple Caching Framework (Haskell)

```haskell
example1 =
  let c0 = makeCache (\n -> n + 1) in
  let (x,c1) = apCache c0 4 in
  let (y,c2) = apCache c1 4 in
  x == y  -- True
```

Fig. 2. Cached Integer Function (Haskell)

```haskell
example2 =
  let c0 = makeCache (\f -> f $ f 0) in
  let inc = \n -> n + 1 in
  let (x,c1) = apCache c0 inc in
  -- ^^ Type error: no Ord for function
  ...
```

Fig. 3. Functional Caching Failure

```haskell
data Symbol = Inc | Plus Int | Twice Symbol
  deriving (Eq, Ord)
example3 =
  let be Inc = \n -> n + 1
      be (Plus k) = \n -> k + n
      be (Twice f) = \n -> be f $ be f $ n in
  let c0 = makeCache (\f -> be f $ be f $ 0) in
  let (x,c1) = apCache c0 Inc in  -- 2
  let (y,c2) = apCache c1 Inc in  -- 2
  let (z,c3) = apCache c2 (Twice Inc) in -- 4
  x == y  -- True
```

Fig. 4a. Defunctionalized Caching (Haskell)

```haskell
example4 =
  let inc = \%Ord n -> n + 1 in
  let plus = \%Ord k n -> k + n in
  let twice = \%Ord f n -> f %$ f %$ n in
  let c0 = makeCache (\f -> f %$ f %$ 0) in
  let (x,c1) = apCache c0 inc in  -- 2
  let (y,c2) = apCache c1 inc in  -- 2
  let (z,c3) = apCache c2 (twice %$ inc) in -- 4
  x == y  -- True
```

Fig. 4b. Caching Intensional Functions (Haskell+`ItsFn`)

- a discussion of the implementation of the `IntensionalFunctions` GHC extension as well as a program analysis artifact written using it (Section 6).

We discuss related work in Section 7 and conclude in Section 8.

2 Intensional Functions

This section illustrates intensional functions by example. We contrast how the caching of functions is accomplished via defunctionalization and via intensional functions. We then illustrate the properties of intensional functions and how operations are defined on them. Unless otherwise indicated, these examples can be compiled using our GHC extension, `IntensionalFunctions`, which we discuss in Section 6.

2.1 Defunctionalization by Example

Consider the Haskell code in Figure 1, which implements a generic caching mechanism for functions. A value of type `Cache a b` is a function together with a dictionary which maps the function's domain values `a` to codomain values `b`. Figure 2 illustrates how this code might be used. Crucially, the domain of the function to be cached is constrained to be orderable; this is a requirement of the dictionary storing the cached values. This otherwise-generic caching mechanism is thus inapplicable to higher-order functions, which lack an `Ord` instance, as exemplified in Figure 3.

A canonical approximation of function comparison is defunctionalization. We define a data type identifying each function in our problem domain and use that data type in lieu of the original function. We also define a dispatch function which can recover each original function's behavior

from this data type. In Figure 4a, for instance, the increment function from Figure 3 has been replaced by the `Inc` constructor from the `Symbol` data type. The `be` function recovers the behavior represented by a `Symbol`. As `Symbol` is a first-order data type, it admits an `Ord` instance.

Defunctionalization imposes two significant burdens on the programmer. First: all call sites which previously invoked an implicated function must now be modified to translate the defunctionalized symbol. The `f` function symbol on line 7 of Figure 4a must be translated before it can be called. This transformation can be far-reaching: any function which *might* reach a transformed call site must itself be represented by a defunctionalized symbol, so its call sites must be transformed, and so on.

Second: the environments of partially-applied functions must be enumerated. The `Symbol` type in Figure 4a represents defunctionalized functions of type `Int -> Int`. A partially applied addition, such as `(\k n -> k + n) 4`, can be represented as `Plus 4` using the `Plus` constructor of `Symbol` on line 1 of Figure 4a. Note that the translation of `Plus` on line 5 must acknowledge the difference between non-local values captured in closure (here, `k`) and parameters that are expected to be applied after translation (here, `n`). This is also reflected in the definition of `Symbol` on line 1, where the types of those closure-captured values (here, `Int`) must be enumerated.

This enumeration becomes especially tedious when a defunctionalized function's environment itself contains a function, as this requires the recursive defunctionalization of e.g. the `Symbol` type itself. That is, defunctionalization must be *deep*: functions can refer to non-local function values, so function environments must be defunctionalized as well. In Figure 4a, the `Twice` constructor carries a `Symbol` in lieu of the `Int -> Int` value that represents our actual intent.

Both CloudHaskell and Scala Spores provide language-level support for ameliorating these burdens when serializing functions for transmission to other processes while introducing minimal syntactic overhead within their respective languages. These systems are typed insofar as they can ensure safe serialization of closure-captured values. (CloudHaskell, for instance, produces a type error if closure-captured values are not statically defined.) However, these systems are limited to the task of serialization; programmers cannot access typed representations of closure-captured values. We next illustrate how intensional functions allow programmers to choose type constraints for closure-captured values and use this information to operate on functions.

2.2 Intensional Functions by Example

The problem in Figure 3 is that the argument passed to `apCache`, a function, does not have an `Ord` instance. Defunctionalization replaces this function with first-order data. We present an alternative: defining a form of function whose properties can be inspected to provide the same constraint-satisfying behavior (such as `Ord`) without closure type enumeration or definitional boilerplate.

In Haskell+`ItsFn` — Haskell with our `IntensionalFunctions` extension enabled — the syntax of intensional functions differs from that of extensional functions in two ways. An intensional function starts with the symbol `\%` (rather than `\`); it also requires a *constraint function* before the list of parameters. A constraint function is a type of kind `Type -> Constraint`, such as `Eq` or `Ord` [Bolingbroke 2011]. This constraint function is both positive and negative: all values in closure must conform to it, but the resulting intensional function is guaranteed to conform to it as well. For instance, `\%Eq x -> (x,z)` represents an intensional function for constructing a tuple using its parameter and a non-local variable `z`. The `Eq` here indicates that the type of `z` must conform to `Eq`, but it also guarantees that the type of *the function itself* conforms to `Eq`. Thus, `let f = \%Eq x -> (x,z) in f == f` typechecks (and evaluates to `True`) as long as `Eq z` holds true. Intensional functions are applied using the `%@` (left-associative) and `%$` (right-associative) application operators. For instance, `(\%Eq x -> x + 1) %@ 3` evaluates to `4`. Application does not make use of intensionality.

Momentarily setting aside how intensional functions satisfy their constraint functions, Figure 4b illustrates how we can use these intensional functions to address the problem presented in Figure 3.

The `inc` function defined on line 2 of Figure 4b is an intensional increment function which conforms to `Ord`; it is therefore a suitable argument to the `apCache` function. This `Ord` instance allows `apCache` to recognize `inc` in the second call on line 7 and retrieve its associated value from the cache.

2.3 Comparing Intensional Functions

We now examine how intensional functions satisfy their constraint functions. Recall from above that the constraint function of an intensional function must be satisfied by all values captured in its closure; for instance, an `Eq` intensional function requires that all values it captures in closure satisfy `Eq`. We can use this information about the intensional function's closure to provide an `Eq` definition for the intensional function itself.

At runtime, extensional functions can only be examined in terms of the behavior exhibited by their sole eliminator: application. Intensional functions, by contrast, have *three* eliminators: application, *identification*, and *inspection*. The latter two eliminators yield the program point at which the function was defined and the environment it captured in closure, respectively.

We define an approximation of equality on intensional functions by comparing the program points and environments of these functions for equality as in Figure 5. The type `a ->%Eq b` refers to an intensional function with domain `a`, codomain `b`, and constraint function `Eq`. The type produced by `itsIdentify` contains entirely first-order data, allowing an `Eq` instance to be defined. `itsInspect` produces a list of GADT wrappers, each carrying a proof that its contents are `Eq`. The `Eq` instance for an intensional function produces `true` if the identities and closures of the two functions are equal.

We prove correct this form of *conservative equality* — that intensional functions which are considered equal will always have the same behavior — in Section 5.3. We focus on conservative equality and comparison here for illustration, but the set of constraint functions which may be implemented for intensional functions is open-ended. A `Hashable` implementation, for instance, would follow the same pattern as `Eq` and allow intensional functions to be used as keys in a hashtable.

```
1 instance Eq (a ->%Eq b) where
2   f == g  =  itsIdentify f == itsIdentify g &&
3              itsInspect f == itsInspect g
```

Fig. 5. Intensional Function Equality

```
1 itsEqConst :: forall a b. (Typeable a, Eq a)
2             => a ->%Eq b ->%Eq a
3 itsEqConst = \%Eq x y -> x
4
5 itsConst :: forall c a b. (Typeable a, c a)
6          => a ->%c b ->%c a
7 itsConst = \%c x y -> x
```

Fig. 6. Intensional Function Polymorphism

2.4 Polymorphism

The above examples are monomorphic for simplicity, but polymorphism is possible with both traditional defunctionalization (e.g. via GADT function symbols [Pottier and Gauthier 2004, 2006a]) and intensional functions. In addition to polymorphism of the domain and codomain, intensional functions must contend with polymorphism of constraint functions and closures.

2.4.1. Parametric Polymorphism

Polymorphism on the domain and codomain of an intensional function is relatively straightforward. Consider applying the intensional constant value function `itsEqConst` as defined on line 3 of Figure 6. The application of this function, e.g. `itsEqConst %@ "A"`, works in the same fashion as its extensional counterpart: the type of `itsEqConst` is instantiated and a newly-created type variable is unified with the type of the argument `"A"`. Thus, this expression has type `b ->%Eq String`. The type signature of `itsEqConst`, however, deserves some attention.

The key difference between this intensional application and its extensional equivalent `const "A"` is that the argument of `itsEqConst` is captured in closure. As a result, the intensional function

```
1 longerThan :: forall a. (Typeable a, Eq a)
2            => [a] ->%Eq Int ->%Eq Bool
3 longerThan = \%Eq xs n -> length xs > n
4
5 example :: Bool
6 example = (longerThan %@ ["A"]) == (longerThan %@ [4])
```

Fig. 7. Comparing Intensional Functions

```
1 example :: forall a. (Typeable a, Eq a)
2         => Bool ->%Eq a ->%Eq a ->%Eq a
3 example = \%Eq b ->
4   let f :: forall b. (Typeable b, Eq b)
5         => b ->%Eq b ->%Eq b
6       f = \%Eq x y -> if b then x else y
7   in \%Eq x y -> f %@ y %@ x
```

Fig. 8. Polymorphic Closure Type Error

requires it to conform to the `Eq` constraint function. (It must also be `Typeable`, ensuring a runtime representation of its type. We discuss this requirement in Section 2.4.2.) We must therefore bound the type parameter `a` to ensure that it meets this requirement. We are not required to prove `Eq b`, however, because no values of type `b` are captured in closure: once the `b` value is supplied, the function's body is executed.[1]

Figure 6 also illustrates polymorphism in the constraint function of an intensional function. The definition of `itsConst` generalizes `itsEqConst` to work with any constraint function `c`. Any constraint function may be applied to `itsConst` either explicitly via type application (as in `itsConst @Eq`) or by type inference. The closure-captured argument must conform to `c` (thus the `(c a)` precondition), but no other special handling is required. This is a natural consequence of the `ConstraintKinds` language extension which was introduced to GHC in version 7.4 [Bolingbroke 2011].

2.4.2. Typeable Environments

In addition to conforming to the constraint function specified by the intensional function, any closure-captured values must also be `Typeable`. To see why, consider the example in Figure 7. On line 6, both sides of the comparison have the type `Int ->%Eq Bool`. Both `["A"]` and `[4]` are captured in their respective closures and have instances for `Eq`. Nonetheless, the environments of these functions are not comparable to *each other*. More generally, this situation arises when a polymorphic intensional function captures a value in closure whose type (a) contains an instantiated type variable and (b) is no longer represented in the resulting function type.

To resolve this issue, we require `Typeable` of all values captured in closure. When two closures are e.g. compared for equality, their types are checked at runtime. In Figure 7, for instance, `example` evaluates to `False`: we do not take two functions with differently-typed environments to be equal.[2]

Rejecting environment polymorphism Although the domain, codomain, and constraint function of an intensional function may be polymorphic, closure-captured values may not. This restriction is a consequence of limitations in GHC's type system and is a weakness of intensional functions in comparison to their extensional equivalents. Thankfully, this is not a common problem in practice.

This monomorphic closure restriction is illustrated by the convoluted code in Figure 8, which does not typecheck. The type error arises on line 7 where `f`, which is polymorphic, is captured in the closure of the anonymous function. GHC typechecks the analogous extensional code.

We are unconcerned about this limitation for two reasons. The first is that, even if polymorphic local bindings could be captured in the closure of an intensional function, there would be no effective way to satisfy the intensional function's constraint function due to a fundamental limitation of GHC's type system. In Figure 8, for instance, we would require an `Eq` instance for the (polymorphic)

[1] The constraint `Eq a` is due to the *possibility* that a value of type `a` is captured in closure. For usability, our implementation imposes such constraints only *at call sites* where such closures are actually built. We discuss this in Section 6.2.

[2] Constraint functions defining only unary operators (such as `Hashable`) shouldn't require `Typeable`, but composition of constraint functions is non-trivial as of GHC version 9.2. We require `Typeable` of all closure-captured values for ease of use.

```
combineSpans :: Search ()
combineSpans = intensional Ord do
  (x,i,j) <- lookup AllSpans ()
  (y,k) <- lookup SpansStartingAt j
  z <- lookup GrammarCombining (x,y)
  insert AllSpans () (z,i,k)
  insert SpansStartingAt i (z,k)
```

Fig. 9a. Intensional Search (Sugared)

```
combineSpans :: Search ()
combineSpans =
  itsBind (lookup AllSpans ()) %$ \%Ord (x,i,j) ->
  itsBind (lookup SpansStartingAt j) %$ \%Ord (y,k) ->
  itsBind (lookup GrammarCombining (x,y)) %$ \%Ord z ->
  itsBind (insert AllSpans () (z,i,k)) %$ \%Ord () ->
  insert SpansStartingAt i (z,k)
```

Fig. 9b. Intensional Search (Desugared)

type of `f`. GHC does not presently permit typeclass instances for polymorphic types because inferring uses of such typeclass instances is extremely difficult [Serrano et al. 2020, 2018].

Our second reason for being unconcerned about this limitation is more practical. The authors of the OutsideIn(X) type system [Vytiniotis et al. 2011] demonstrated that local polymorphic let bindings are uncommon in practice. In that work, the authors reported that fewer than 4% of the modules in GHC's standard libraries relied upon local polymorphic bindings and fewer than 12% of Hackage packages had any modules which did so. This suggests that actual instances of this problematic example would be rare in practice. In the rare event that the need arose, a simple workaround exists: to pack the polymorphic type in a `newtype` using `RankNTypes`.

3 Intensional Monads

The introduction of intensional functions prompts us to consider the myriad ways in which extensional functions are used and to investigate their intensional analogues. Herein, we briefly discuss one such example: *intensional monads*, a reconstruction of Haskell's encoding of monads with intensional Kleisli functions.[3] While the signature of a traditional monad bind operator is `bind :: m a -> (a -> m b) -> m b`, the intensional form is `itsBind :: m a ->%c (a ->%c m b) ->%c m b` for a particular constraint function `c`.

Note that the closure of the bound intensional function `a ->%c m b` must conform to `c`, so the `itsBind` implementation may make use of this guarantee. As an example, we consider early pruning within an *idempotent* search: a search in which we are concerned only with results and not how we arrived at them. Consider a `Search` monad equipped with some related `Index` type constructor and two operations: `lookup`, which produces each entry in an `Index` for a given key, and `insert`, which adds an entry to an `Index`. Let us assume the following type signatures:

```
lookup :: (Ord (Index k v), Ord k, Ord v) => Index k v -> k -> Search v
insert :: (Ord (Index k v), Ord k, Ord v) => Index k v -> k -> v -> Search ()
```

Crucially, we expect each an operation bound to a `lookup` (that is, the `f` in each `itsBind (lookup i k) f`) to run for each associated value in the `Index`, *even those which are added in the future.*

We briefly describe how such a `Search` monad might work. The monad can deduplicate redundant calls to `insert` by encapsulating a dictionary data structure to hold indexed values. As new indices are added to this dictionary, the monad is obligated to pass them to previous `lookup` operations as mentioned above. To do this, the monad "catches up" by re-evaluating previous computations dependent upon that index by passing them the new index value. To track these computations, the monad must store each continuation (the `f` in `itsBind (lookup i k) f`). In an extensional monad, these continuations `f` are extensional and so cannot be examined or readily deduplicated.

An intensional `Search` monad resolves this issue: the continuations passed to `itsBind` are intensional functions and, if subject to the `Ord` constraint, can be compared and deduplicated like any first-order value. For instance, consider the program fragment appearing in Figure 9a and its

[3] A more thorough examination of intensional monads appears in this paper's supplementary material.

desugared counterpart in Figure 9b.[4] On line 5 of each, the remainder of the algorithm runs for each `z` value associated with the key `(x,y)` in the index `GrammarCombining`.

While deduplicating `z` is straightforward in any implementation, an intensional `Search` permits us to *deduplicate the continuation* `\%Ord z -> ...` based upon the values of `i` and `k` it has captured in closure: two continuations with the same `i` and `k` are equal according to the approximation of Section 2.3. The value of `j`, which is no longer relevant at this point, is naturally excluded from this deduplication process because it is not captured in the continuation's closure. Observe that the continuations (i.e., the second arguments) passed to `itsBind` tend themselves to capture `itsBind` in closure; as a result, it is critical that `itsBind`, and so the monad itself, is intensional.

The next two sections provide a formal treatment of intensional functions in support of this and other use cases.

4 Lazy Substitution

This section introduces λ_θ, a small lambda calculus which uses *lazy substitution*. This prepares us to introduce in Section 5 the intensional functions lambda calculus, λ_{ITS}, which also uses lazy substitution. A lambda calculus using lazy substitution is equivalent to a lambda calculus using traditional substitution, but lazy substitution considerably simplifies some λ_{ITS}-related proofs.

We discuss in Section 7 some work related to lazy substitution (such as explicit substitution [Abadi et al. 1990]). This section defines λ_θ to introduce lazy substitution separately from the details of intensional functions. Key to lazy substitution is that, while substitutions are manifest as a part of the grammar, they are not a form of expression.

4.1 Defining λ_θ

We define the syntax of λ_θ in Figure 10. λ_θ is a call-by-name lambda calculus in which substitutions θ – sequences of mappings from variables to expressions – appear as components of the grammar. By representing these typical capture-avoiding substitution operations explicitly, we are able to simplify proofs of properties about the effects of substitutions on evaluation.

$$\begin{array}{lcll} e & ::= & x \mid \lambda_\theta x.\, e \mid e\ e & \textit{expressions} \\ \theta & ::= & [x \mapsto e, \ldots] & \textit{substitutions} \end{array}$$

Fig. 10. λ_θ Syntax

To discuss the syntax in this figure, we require some basic notation:

Definition 4.1. We write $\textsc{fv}(e)$ to denote the free variables of e and $\textsc{dom}(\theta)$ for $\{x \mid x \mapsto e \in \theta\}$.

Substitutions are formally defined as a list of mappings from variable to expression. We define the substitution operation in Figure 11, overloading mathematical function notation. This is a typical capture-avoiding substitution definition except that (1) it performs each of a *list* of substitutions and (2) *substitutions stop at lambda abstractions*. Upon reaching a lambda abstraction, the substitution to be performed is stored in the θ position of the lambda rather than being applied directly to the body. This is the sense in which substitution is "lazy": we will not perform substitution until it is required to continue reduction.

We define the call-by-name operational semantics for λ_θ in Definition 4.2. The Appl rule performs on the function's body any substitutions which were previously deferred (in addition to the substitution of the parameter). These rules do not allow evaluation under binders; we make this choice to simplify our statements below.

[4]Some readers may recognize this as a rule from CKY chart parsing [Cocke 1969; Kasami 1965; Sakai 1961; Younger 1967].

$$
\begin{array}{rcll}
[](e') & = & e' & \\
([x \mapsto e] \parallel \theta)(x) & = & \theta(e) & \\
([x \mapsto e] \parallel \theta)(x') & = & \theta(x') & , x \neq x' \\
([x \mapsto e] \parallel \theta)(\lambda_{\theta'} x.\, e') & = & \theta(\lambda_{\theta'} x.\, e') & \\
([x \mapsto e] \parallel \theta)(\lambda_{\theta'} x'.\, e') & = & \theta(\lambda_{(\theta' \parallel [x \mapsto e])} x'.\, e') & , x \neq x', x' \notin \textsc{fv}(e) \\
\theta(e_1\ e_2) & = & \theta(e_1)\ \theta(e_2) &
\end{array}
$$

Fig. 11. λ_θ Substitution

$$
\textsc{Red-Left}\ \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad\qquad \textsc{Appl}\ \frac{e' = (\theta \parallel [x \mapsto e_2])(e_1)}{(\lambda_\theta x.\, e_1)\ e_2 \longrightarrow e'}
$$

Fig. 12. λ_θ Operational Semantics

Definition 4.2. Let $e \longrightarrow e'$ be the least relation satisfying the rules in Figure 12 as well as traditional α-renaming. Let $e \longrightarrow^* e'$ be the transitive closure of this relation.

In the next section, we discuss some formal properties of λ_θ. We will use these same properties in the larger setting of the λ_{ITS} system defined in Section 5, but the relative simplicity of λ_θ correspondingly simplifies the initial presentation of these properties.

4.2 Formal Properties of λ_θ

We note that a bisimulation exists which relates an expression's evaluation in λ_θ to its evaluation in a traditional call-by-name lambda calculus: at each step, suspended substitutions in the λ_θ expression can be eagerly performed to produce the traditional expression. (We elide formal definitions and proof for brevity.) We also observe that the set of unique function bodies in λ_θ expressions is *nonincreasing* as evaluation proceeds. Formally:

Lemma 4.3. Suppose $e_1 \longrightarrow e_2$. Let $E_k = \{e \mid (\lambda_\theta x.\, e) \text{ appears in } e_k\}$ for $k \in \{1, 2\}$. Then $E_1 \supseteq E_2$.

Proof. By induction first on the height of $e_1 \longrightarrow e_2$ and then on the height of the substitution applied in the Appl rule. In summary: substitution does not modify the bodies of functions and the Appl rule removes the applied function's (substituted) body from its surrounding function. □

Note that Lemma 4.3 relies upon the fact that λ_θ does not permit evaluation under lambdas. Any such evaluation would modify the body of a function and thus break this property.

While Lemma 4.3 demonstrates that function *bodies* are nonincreasing as evaluation proceeds, this is not true of functions themselves. As an expression evaluates, new $\lambda_\theta x.\, e$ subexpressions appear with variations in the θ position. Intuitively, θ is an environment: its substitutions correspond to bindings for the function body's non-local variables. This illustrates our interest in the λ_θ-calculus: functions are explicitly defined in terms of their original definition in the source program and the values captured in their closure. Lemma 4.3 is a common invariant of functional compilation systems: new *environments* appear at runtime but new *code* does not.

Our overall goal is to show the (conservative) equality of functions with the same environment. Substitutions act as environments but require a notion of equivalence. For instance, the substitutions $[x_1 \mapsto (\lambda_{[]} x_0.\, x_0)]$ and $[x_1 \mapsto x_2, x_2 \mapsto (\lambda_{[]} x_0.\, x_0)]$ are grammatically distinct entities but will, when applied, always have the same results. Intuitively, two substitutions are equivalent if, for all input expressions, they produce substitution-equivalent expressions; substitution equivalence on expressions is a homomorphic extension of this definition. Formally:

Definition 4.4. We mutually define relations of the forms $\theta \simeq \theta$ and $e \simeq e$ as the least relations conforming to the following:

$$\theta_1 \simeq \theta_2 \quad \text{if } \forall e.\, \theta_1(e) \simeq \theta_2(e) \qquad\qquad x \simeq x$$
$$e_1\ e_2 \simeq e_1'\ e_2' \quad \text{if } e_1 \simeq e_1' \text{ and } e_2 \simeq e_2' \qquad\qquad \lambda_{\theta_1} x.\, e_1 \simeq \lambda_{\theta_2} x.\, e_2 \quad \text{if } \theta_1 \simeq \theta_2 \text{ and } e_1 \simeq e_2$$

Substitution equivalence allows us to make observations about substitutions in λ_θ. For instance, substitutions only affect the free variables in the expressions to which they are applied. (This is intuitive from inspection of Definition 11 but is crucial in our later proof and so deserves formal treatment.) Let us denote sets of variables as X. Then, in keeping with our view of substitutions as functions on expressions, let us define a restriction of substitutions to a specific set of variables.

Definition 4.5.
$$\begin{array}{rcll} []|_X & = & [] & \\ ([x \mapsto e] \parallel \theta)|_X & = & \theta|_X & , x \notin X \\ ([x \mapsto e] \parallel \theta)|_X & = & [x \mapsto e] \parallel (\theta|_{X \cup \mathrm{FV}(e)}) & , x \in X \end{array}$$

Definition 4.5 filters a substitution by discarding mappings for variables not in the set of approved variables. We must be careful to preserve substitutions for variables which will be introduced by other substitutions. Using this notation, we can formally state the above claim:

Lemma 4.6. For any substitution θ and any expression e, $\theta(e) \simeq \theta|_{\mathrm{FV}(e)}(e)$.

Proof. By induction on the length of θ, then the height of e, then case analysis of Definition 11. □

We also observe that substitutions which produce equivalent results on a particular expression will produce equivalent results on all expressions with the same free variables. Formally:

Lemma 4.7. If $\theta_1(e) \simeq \theta_2(e)$ then $(\theta_1|_{\mathrm{FV}(e)}) \simeq (\theta_2|_{\mathrm{FV}(e)})$.

Proof. By induction on the height of e and by using Figure 11 and $\theta_1(e)$ to infer the substitutions performed by θ_1. In summary: we view e as a template structure holding free variables. Substitution is homomorphic except on variables (which are immediately replaced) and functions (which store the substitution directly in their θ position). This does not allow us to infer the exact θ_1 or θ_2 — different substitutions may yield the same results on the free variables of e — but we learn enough to determine how the substitutions behave on those free variables. □

In the following sections, we will define and prove formal properties about the larger λ_{ITS} system which includes intensional functions. While the grammar of λ_{ITS} is much larger, these same arguments regarding λ_θ still apply to λ_{ITS}.

5 Formalization of Intensional Functions

This section defines λ_{ITS}, a lambda calculus equipped with intensional functions and the features to make them meaningful, and examines the formal properties of that system. Most importantly, we prove the correctness of the intuitive conservative function equality model we presented in Section 2.3. We also give a type system in Section 5.4 which is proven correct in this paper's supplemental material. Section 5.5 illustrates an example of a small λ_{ITS} program.

5.1 λ_{ITS} Features

The grammar of λ_{ITS} appears in Figure 13. We now briefly motivate each of its features.

λ_{ITS} includes constraint functions F which name single-variable polymorphic types. We use constraint functions to represent properties we wish to define on intensional functions; for instance, the type $\texttt{bool} \xrightarrow{\mathsf{Eq}} \texttt{bool}$ denotes intensional functions (from booleans to booleans) which are equatable. Constraint functions are defined using the `class` keyword as they correspond to a weak form of typeclass. A reader may safely think of these constraint functions as single-method typeclasses which conflate the name of the method with the name of the class. For instance, a λ_{ITS} program

x	*variables*	$v ::= x \mid \ell \mid F \mid \phi \mid e :: e \mid \texttt{nil}^{\tau} \mid \texttt{true} \mid \texttt{false} \mid$	*values*
ℓ	*program points*	$\quad \texttt{tyrep}\ \tau \mid \lambda^{F}_{\ell,e,\theta} x{:}\tau.\, e \mid \Lambda\alpha.\, Q \Rightarrow e$	
F	*constraint functions*	$e ::= v \mid e\ e \mid e\ \tau \mid \texttt{identify}\ e \mid \texttt{inspect}\ e \mid$	*expressions*
$\phi ::= \langle F, e, \tau \rangle$	*closure items*	$\quad \texttt{pack}\ e\ \texttt{as}\ q \mid \texttt{unpack}\ x : \exists\alpha\ \texttt{as}\ x = e\ \texttt{in}\ e \mid$	
$q ::= F\ \tau$	*constraints*	$\quad \texttt{let}\ x{:}\sigma = e\ \texttt{in}\ e \mid e \sim e\ ?\ e : e \mid e == e \mid$	
$Q ::= \{q, \ldots\}$	*constraint sets*	$\quad \texttt{hd}\ e \mid \texttt{tl}\ e \mid \texttt{nil?}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	
$C ::= \{F \mapsto \sigma, \ldots\}$	*constraint names*	$c ::= \texttt{class}\ F : \forall\alpha.\, \tau;$	*class declarations*
$W ::= \{q \mapsto e, \ldots\}$	*constraint witnesses*	$d ::= \texttt{instance}\ q = e;$	*instance declarations*
$\Gamma ::= \{x \mapsto \sigma, \ldots,\}$	*type environments*	$p ::= \overline{c}\ \overline{d}\ e$	*programs*
$\psi ::= x \mapsto e \mid \alpha \mapsto \tau$	*substitutions*	$\tau ::= \alpha \mid \tau \xrightarrow{F} \tau \mid [\tau] \mid \texttt{bool} \mid \texttt{ppt} \mid$	*monotypes*
$\theta ::= [\psi, \ldots]$	*substitution sequences*	$\quad \texttt{clo}\ F \mid \texttt{tyrep}\ \tau$	
		$\sigma ::= \forall\alpha.\, Q \Rightarrow \sigma \mid \tau$	*polytypes*

Fig. 13. λ_{ITS} Grammar

might include `class Eq : ∀a. a` $\xrightarrow{\texttt{Eq}}$ `a` $\xrightarrow{\texttt{Eq}}$ `bool` to designate the type of equality functions. (Equality functions themselves are equatable in this definition. We discuss this further in Section 5.5.)

Constraints are satisfied by ad-hoc instantiations. Constraints q are syntactic pairs of a constraint function and a monotype to satisfy it. Constraint functions appear as terms in the expression grammar to be used via explicit type application to identify a particular instantiation. For instance, a previously-defined equality on booleans may be named as `Eq bool`; the expression `(Eq bool) true false` would evaluate to `false`. In general, constraint functions themselves will be single-variable polymorphic functions with empty type constraint sets.

Intensional functions themselves are written $\lambda^{F}_{\ell,e',\theta} x{:}\tau.\, e$. The three rightmost positions in this form — x, τ, and e — are a parameter, its type, and the function body as in a typical typed lambda calculus. F is the constraint function to which the intensional function conforms. The ℓ in the first lower position corresponds to a unique program point at which the function was originally defined; e' is a closure expression which will, in practice, be a list of type-tagged closure items ϕ. The θ position corresponds to the substitutions described in the λ_θ-calculus in Section 4.

Although λ_{ITS} is much more complex than λ_θ, substitutions operate in the same fashion. For brevity, we omit much of the definition of capture-avoiding substitution for λ_{ITS}, but we give the clauses pertaining to intensional functions (including type substitution) for clarity.

Definition 5.1. We use $\theta(e)$ to denote the lazy capture-avoiding substitution of θ in the expression e; we use similar notation for other grammar terms such as p and τ.

$$
\begin{aligned}
([x' \mapsto e'] \parallel \theta)(\lambda^{F}_{\ell,e'',\theta'} x{:}\tau.\, e) &= \theta(\lambda^{F}_{\ell,e'',\theta'} x{:}\tau.\, e) && , x = x' \\
([x' \mapsto e'] \parallel \theta)(\lambda^{F}_{\ell,e'',\theta'} x{:}\tau.\, e) &= \theta(\lambda^{F}_{\ell,e'',(\theta' \parallel [x' \mapsto e'])} x{:}\tau.\, e) && , x \neq x', x \notin \text{FV}(e') \\
([\alpha' \mapsto \tau'] \parallel \theta)(\lambda^{F}_{\ell,e'',\theta'} x{:}\tau.\, e) &= \theta(\lambda^{F}_{\ell,e'',(\theta' \parallel [\alpha' \mapsto \tau'])} x{:}\tau.\, e) \\
&\vdots
\end{aligned}
$$

Let us consider an example intensional function expression. To ease the presentation of such examples throughout this section, we will use the simple syntactic sugar presented in Figure 14. The function $(\lambda^{\texttt{Eq}}_{1,[],[]} \texttt{a:bool}.\, \lambda^{\texttt{Eq}}_{2,[\texttt{pack a as Eq bool}],[]} \texttt{b:bool.a and b})$ is a two-argument function performing the logical conjunction of its arguments. We denote program points as integers 1 and 2 to distinguish them from other terms. The first function's closure is `[]` as that function's body has no non-local variables. The second function's closure contains a single element, a type-tagged packing of `a`, because `a` is non-local and free in that function's body. The `Eq bool` appearing in that `pack` expression is an annotation for the type system signifying that an instance of `Eq` must exist

for `bool`, the type of `a`. Both functions have [] in their substitution position as neither has yet been subjected to any substitutions during evaluation.

$$\begin{array}{lcl} e_1 \texttt{ and } e_2 & \equiv & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else false} \\ e_1 \texttt{ or } e_2 & \equiv & \texttt{if } e_1 \texttt{ then true else } e_2 \\ [e_1, \ldots, e_n] & \equiv & e_1 :: \ldots :: e_n :: \texttt{nil} \end{array}$$

Fig. 14. λ_{ITS} Syntactic Sugar

We have specific expectations of the λ_{ITS} programs we will consider: program points should be unique at the start of evaluation, closures should capture all (and only) non-local variables of their corresponding functions, and substitutions should be initially empty and accumulate lazy substitution operations over time. However, these expectations cannot be enforced by syntax any more than whether expressions are closed. Section 5.3 will formally define invariants to identify which programs conform to these expectations and are therefore included in our study. In practice, these invariants are enforced by an encoding system (such as Haskell+`ItsFn`) which translates from a higher-level language that does not require the programmer to articulate program points, closures, or substitutions.

As we continue to examine the syntax of λ_{ITS}, recall from Section 2.4: even if we know that two intensional functions' closures are comprised of equatable elements, we must further know that they are equatable *to each other*. λ_{ITS} supports runtime type comparisons using a special form of conditional expression written $e_1 \sim e_2 \texttt{ ? } e_3 \texttt{ : } e_4$. Here, e_1 and e_2 must be runtime type witnesses of the form $\texttt{tyrep } \tau$. This expression reduces to e_3 if they are equal and e_4 if they are not. We briefly discuss the typing of this expression using standard techniques in Section 5.4.

The grammar of λ_{ITS} also supports bounded polymorphism via explicit type application. While polymorphism isn't strictly necessary in λ_{ITS}, we include it to demonstrate its compatibility with intensional functions. Polymorphism is bounded via qualified constraints. [Jones 1992]

5.2 Operational Semantics

We now formalize λ_{ITS} beginning with the operational semantics of expressions. As λ_{ITS} has several expression forms, we abbreviate our definition using evaluation contexts [Felleisen and Hieb 1992] to identify points of reduction. Evaluation contexts are similar to the expression grammar and contain a single "hole", denoted $\bullet$, to indicate the point at which reduction can occur. As our operational semantics are call-by-name, the evaluation contexts defined in Figure 15 are sufficient. We write $\xi(e)$ to denote the expression produced by substituting e for the hole appearing in ξ.

$$\begin{array}{lll} \xi ::= \bullet \mid \xi\ e \mid \xi\ \tau \mid \texttt{identify } \xi \mid \texttt{inspect } \xi \mid \texttt{unpack } x : \exists \alpha \texttt{ as } x = \xi \texttt{ in } e & & \textit{evaluation contexts} \\ \quad \mid \xi \sim e \texttt{ ? } e \texttt{ : } e \mid v \sim \xi \texttt{ ? } e \texttt{ : } e \mid \xi == e \mid v == \xi \mid \texttt{hd } \xi \mid \texttt{tl } \xi \mid \texttt{nil? } \xi \mid \texttt{if } \xi \texttt{ then } e \texttt{ else } e & & \end{array}$$

Fig. 15. λ_{ITS} Evaluation Contexts

Our operational semantics relation is defined in terms of a witness environment W which maps each constraint (e.g. `Eq bool`) to its corresponding definition. We use $W[q]$ to denote the lookup of a constraint's expression in this environment. We now define the operational semantics of λ_{ITS}:

Definition 5.2. We define $W \vdash e \longrightarrow e$ to be the least relation satisfying the rules in Figure 16.

Application of intensional functions is identical to that of extensional functions in λ_θ (which is, as previously mentioned, bisimilar to a traditional call-by-name lambda calculus). Lazy substitution is performed at application and when the closure of a function is obtained via the `inspect` primitive; the `identify` primitive simply retrieves the corresponding program point. λ_{ITS} includes primitives for conditions, lists, and comparing program points for equality.

The E-Witness rule deserves some brief attention. Although the rest of the operational semantics is substitution-based, the invocation of constraint implementations is environment-based. This is due to polymorphism: while variable bindings are immediate from lexical analysis, the connection

$$\textsc{E-Red}\ \frac{W \vdash e \longrightarrow e'}{W \vdash \xi(e) \longrightarrow \xi(e')} \qquad \textsc{E-App}\ \frac{}{W \vdash (\lambda^F_{\ell,e',\theta} x{:}\tau.\, e_1)\ e_2 \longrightarrow (\theta \,\|\, [x \mapsto e_2])(e_1)}$$

$$\textsc{E-TApp}\ \frac{}{W \vdash (\Lambda\alpha.\, Q \Rightarrow e)\ \tau \longrightarrow [\alpha \mapsto \tau](e)} \qquad \textsc{E-Witness}\ \frac{}{W \vdash F\ \tau \longrightarrow W[F\ \tau]}$$

$$\textsc{E-Identify}\ \frac{}{W \vdash \texttt{identify}\ (\lambda^F_{\ell,e',\theta} x{:}\tau.\, e) \longrightarrow \ell} \qquad \textsc{E-Inspect}\ \frac{}{W \vdash \texttt{inspect}\ (\lambda^F_{\ell,e',\theta} x{:}\tau.\, e) \longrightarrow \theta(e')}$$

$$\textsc{E-Pack}\ \frac{}{W \vdash \texttt{pack}\ e\ \texttt{as}\ F\ \tau \longrightarrow \langle F, e, \tau\rangle}$$

$$\textsc{E-Unpack}\ \frac{}{W \vdash \texttt{unpack}\ x_1 : \exists\alpha\ \texttt{as}\ x_2 = \langle F, e, \tau\rangle\ \texttt{in}\ e' \longrightarrow [x_1 \mapsto e, x_2 \mapsto \texttt{tyrep}\ \tau, \alpha \mapsto \tau](e')}$$

$$\textsc{E-Let}\ \frac{}{W \vdash \texttt{let}\ x{:}\sigma = e\ \texttt{in}\ e' \longrightarrow [x \mapsto e](e')}$$

$$\textsc{E-Like}\ \frac{}{W \vdash (\texttt{tyrep}\ \tau \sim \texttt{tyrep}\ \tau\ ?\ e_1 : e_2) \longrightarrow e_1}$$

$$\textsc{E-Unlike}\ \frac{\tau_1 \neq \tau_2}{W \vdash (\texttt{tyrep}\ \tau_1 \sim \texttt{tyrep}\ \tau_2\ ?\ e_1 : e_2) \longrightarrow e_2}$$

(omitted for brevity: rules for `==`, `::`, `nil?`, *and* `if`*)*

Fig. 16. λ_{ITS} Operational Semantics: Expression Evaluation Rules

$$\textsc{P-Step}\ \frac{W = \left\{q \mapsto e \;\middle|\; (\texttt{instance}\ q = e\,;) \in \overline{d}\right\} \qquad W \vdash e \longrightarrow e'}{\overline{c}\ \overline{d}\ e \longrightarrow \overline{c}\ \overline{d}\ e'}$$

Fig. 17. λ_{ITS} Operational Semantics: Program Evaluation

between a constraint definition and its usage may not be apparent until after a type application. There are no occurrences of "`Eq bool`" in the expression "$(\Lambda\alpha.\, \emptyset \Rightarrow \texttt{Eq}\ \alpha)\ \texttt{bool}$", for instance, but application of the E-TApp rule reveals a use of the `Eq bool` constraint. For this reason, we must maintain an environment W of constraints to be consulted once type application is resolved.

We evaluate λ_{ITS} programs simply by packing their witnesses into a static W dictionary and stepping the body expression. We define all constraints at top level to simplify the use of W: because it is global, one static W can be used throughout evaluation. Formally:

Definition 5.3. We define $p \longrightarrow p'$ to be the least relation satisfying the rule in Figure 17. We define $p \longrightarrow^* p'$ to hold iff either $p = p'$ or $p \longrightarrow \ldots \longrightarrow p'$.

5.3 Closure Consistency

We now prove correct the conservative comparison of intensional functions described in Section 2.3. That model equates functions which have the same program point and environment. We will show that, for well-formed λ_{ITS} programs, the program point and environment of each intensional function decide the rest of that function. While this proof is limited to conservative equality, arguments of similar structure can be used to support other operations (e.g. function comparison or hashing).

We begin by establishing some preliminaries:

Definition 5.4. We define $\textsc{fv}(e)$ to be the set of free variables x appearing in e. We define $\textsc{ftv}(\tau)$ to be the set of free type variables α appearing in τ. We extend FTV to operate homomorphically on constraint sets Q, environments Γ, expressions e, and polytypes σ. Let Canon be a function from sets of variables and type variables to a list of those variables in some canonical order.

We then establish a canonical closure representation for λ_{ITS} functions.

Definition 5.5. For a parameter x_0, a body expression e, and a constraint function F, let $[x_1, \ldots, x_n] = \textsc{Canon}(\textsc{fv}(e) \backslash \{x_0\})$ and let $[\alpha_1, \ldots, \alpha_m] = \textsc{Canon}(\textsc{ftv}(e))$. An expression e' is a *canonical closure* of x_0, e, and F iff $e' = [\texttt{pack}\ x_1\ \texttt{as}\ F\ \tau_1\texttt{,}\ \ldots\texttt{,}\ \texttt{pack}\ x_n\ \texttt{as}\ F\ \tau_n\texttt{,}\ \texttt{pack}\ \texttt{(tyrep}\ \alpha_1\texttt{)}\ \texttt{as}\ F\ \texttt{(tyrep}\ \alpha_1\texttt{)}\texttt{,}\ \ldots\texttt{,}\ \texttt{pack}\ \texttt{(tyrep}\ \alpha_m\texttt{)}\ \texttt{as}\ F\ \texttt{(tyrep}\ \alpha_m\texttt{)}]$ for some $\tau_1, \ldots, \tau_n$.

Note that Definition 5.5 describes *a* canonical closure, not *the* canonical closure, of the defining lexical components of a function. This is because the monotypes $\tau_1, \ldots, \tau_n$ are not lexically fixed by these lexical components. We discuss typechecking of λ_{ITS} in Section 5.4 and, when typechecking, only one selection of these monotypes will produce a canonical typing of the program. For the purposes of the proofs in this section, however, we need not constrain these monotypes.

We now define *closure consistency*, the property we aim to show of well-formed programs which will support our conservative approximation of function equality.

Definition 5.6. An expression e is *closure consistent* iff the following are true:

(1) For every function $\lambda^F_{\ell,e_1,\theta} x : \tau \,.\, e_2$ appearing in e,
  (a) e_1 is a canonical closure of x, e_2, and F.
  (b) $x \notin \textsc{dom}(\theta)$.
(2) For every pair of functions $\lambda^{F_1}_{\ell,e'_1,\theta_1} x_1 : \tau_1 \,.\, e''_1$ and $\lambda^{F_2}_{\ell,e'_2,\theta_2} x_2 : \tau_2 \,.\, e''_2$ (note same ℓ!) in e,
  (a) $F_1 = F_2$, $e'_1 = e'_2$, $x_1 = x_2$, $\tau_1 = \tau_2$, and $e''_1 = e''_2$. (That is: any two functions with the same program point differ only by substitutions.)
  (b) $\theta_1(e'_1) \simeq \theta_2(e'_2)$ implies $\theta_1(e''_1) \simeq \theta_2(e''_2)$. (That is: any two functions with the same program point and equivalent substitutions will produce equivalent bodies after applying their substitutions.)

We extend the definition of closure consistency homomorphically to programs p and constraint witnesses W.

Closure consistency allows us to reason about functions in terms of their program points and environments rather than substitutions on their bodies. By including lazy substitution, we can reason instead about program points and *substitutions*. As a consequence, we can use properties similar to those shown in Section 4.2 to validate our conservative equality model.

We thus aim to preserve closure consistency throughout evaluation. However, many λ_{ITS} programs are not closure consistent. For example, the program "$[\lambda^{\texttt{Eq}}_{1,[],[]}\texttt{a:bool.true}, \lambda^{\texttt{Eq}}_{1,[],[]}\texttt{a:bool.false}]$" contains two functions with the same program points and environments but different bodies.

We proceed by defining a set of *initial* programs which meet this closure consistency property and then showing preservation of closure consistency among those programs during evaluation. As mentioned above, we expect programmers to write in a higher-level language (e.g. Haskell+`ItsFn`) which only requires (and only permits) the programmer to specify F. The encoding should then establish functions' program points and environments. We define initial programs as follows:

Definition 5.7. A program p is *initial* iff the following are true:

(1) For every function $\lambda^F_{\ell,e_1,\theta} x : \tau \,.\, e_2$ appearing in p, e_1 is a canonical closure of x, e_2, and F.
(2) For every function $\lambda^F_{\ell,e_1,\theta} x : \tau \,.\, e_2$ appearing in p, $\theta = []$.

(3) For every pair of functions $\lambda^{F_1}_{\ell_1,e',\theta_1} x_1 : \tau_1 \,.\, e_1''$ and $\lambda^{F_2}_{\ell_2,e',\theta_2} x_2 : \tau_2 \,.\, e_2''$ appearing in p, we have $\ell_1 \neq \ell_2$. (That is: no two functions have the same program point.)

To establish a starting point, we show that these programs are closure consistent:

Lemma 5.8. Any initial program p is closure consistent.

Proof. By clauses 1 and 2 of Definition 5.7, all functions in p have canonical closure expressions and empty substitutions; this satisfies clauses 1a and 1b of Definition 5.6. By clause 3 of Definition 5.7, all functions in p have distinct program points; this satisfies clauses 2a and 2b of Definition 5.6. □

Lemma 5.8 is our base case for proving that initial programs are closure consistent throughout evaluation. We next prove that closure consistency is preserved as evaluation proceeds, writing one lemma for each clause of Definition 5.6. First, we show that canonical closures are preserved.

Lemma 5.9. If p is closure consistent and $p \longrightarrow p'$ then, for every function $\lambda^{F}_{\ell,e_1,\theta} x : \tau \,.\, e_2$ appearing in p', $e_1 = \theta(e_3)$ where e_3 is a canonical closure of x, e_2, and F.

Proof. For any function appearing in p', it either appears in p or it does not. In the former case, our goal is immediately satisfied by Definition 5.6 and because p is closure consistent.

In the latter case, we observe by inspection of the rules in Figures 16 and 17 that any function appearing in p' which does not appear in p is the result of substitution of a function appearing in p. By this observation and Definition 5.1, some function $\lambda^{F}_{\ell,e_1',\theta'} x : \tau \,.\, e_2$ appears in p such that $e_1 = \theta''(e_1')$ and $\theta = \theta' \parallel \theta''$. (Note that θ'' is not necessarily the same as a substitution appearing in an operational semantics rule because θ'', by Definition 5.1, excludes all substitutions of x.)

Because p is closure consistent, we have that $e_1' = \theta'(e_3)$ for some e_3 which is a canonical closure of x, e_2, and F. We have $e_1 = \theta''(e_1')$, so $e_1 = \theta''(\theta'(e_3))$. By Definition 5.1 and because $\theta = \theta' \parallel \theta''$, we have $e_1 = \theta(e_3)$, so we are finished. □

We next show the preservation of the second clause of Definition 5.6: functions' parameters do not appear in the domains of functions' substitutions. We use a similar strategy to the previous lemma, using the fact that new functions appear as substitutions of previously-existing functions.

Lemma 5.10. If p is closure consistent and $p \longrightarrow p'$ then, for every function $\lambda^{F}_{\ell,e_1,\theta} x : \tau \,.\, e_2$ appearing in p', $x \notin \textsc{dom}(\theta)$.

Proof. For any function appearing in p', it either appears in p or it does not. In the former case, our goal is immediately satisfied by Definition 5.6 and because p is closure consistent.

In the latter case, consider a function $\lambda^{F}_{\ell,e_1',\theta'} x : \tau \,.\, e_2$ appearing in p' but not appearing in p. By inspection of the rules in Figures 16 and 17, this function is the result of substitution of a function appearing in p. Because p is closure consistent, the function upon which that substitution is performed does not contain x in the domain of its substitution. By Definition 5.1, x is not introduced to the domain of the substitution. Thus, $x \notin \textsc{dom}(\theta)$. □

The next clauses of Definition 5.6 involve pairs of functions, so we extend the previous strategy accordingly. The cases in which *either* or *both* functions are new conveniently generalize.

Lemma 5.11. If p is closure consistent and $p \longrightarrow p'$ then, for every pair of functions $\lambda^{F_1}_{\ell,e_1',\theta_1} x_1 : \tau_1 \,.\, e_1''$ and $\lambda^{F_2}_{\ell,e_2',\theta_2} x_2 : \tau_2 \,.\, e_2''$ appearing in p', we have $F_1 = F_2$, $x_1 = x_2$, $\tau_1 = \tau_2$, and $e_1'' = e_2''$.

Proof. For any function appearing in p', it either appears in p or it does not. We thus have three cases: either both functions appear in p, only one function appears in p, or neither function appears in p. In the first case, we are immediately finished because p is closure consistent. By inspection of

the rules in Figures 16 and 17, any function appearing in p' which does not appear in p is the result of substitution of a function appearing in p.

Consider the case in which neither function appears in p. By the above observation, there exists some function $\lambda^{F_3}_{\ell_3,e'_3,\theta_3} x_3 : \tau_3 \,.\, e''_3$ which appears in p such that $\theta''_3(\lambda^{F_3}_{\ell_3,e'_3,\theta_3} x_3 : \tau_3 \,.\, e''_3) = \lambda^{F_1}_{\ell,e'_1,\theta_1} x_1 : \tau_1 \,.\, e''_1$ for some substitutions θ''_3. Similarly, there exists some function $\lambda^{F_4}_{\ell_4,e'_4,\theta_4} x_4 : \tau_4 \,.\, e''_4$ which appears in p such that $\theta''_4(\lambda^{F_4}_{\ell_4,e'_4,\theta_4} x_4 : \tau_4 \,.\, e''_4) = \lambda^{F_2}_{\ell,e'_2,\theta_2} x_2 : \tau_2 \,.\, e''_2$ for some substitutions θ''_4. By Definition 5.1, we have $F_1 = F_3$, $e'_1 = e'_3$, $x_1 = x_3$, $\tau_1 = \tau_3$, $e''_1 = e''_3$, and $\ell = \ell_3$. Similarly, we have $F_2 = F_4$, $e'_2 = e'_4$, $x_2 = x_4$, $\tau_2 = \tau_4$, $e''_2 = e''_4$, and $\ell = \ell_4$. Since $\ell_3 = \ell = \ell_4$ and p is closure consistent, we have $F_3 = F_4$, $x_3 = x_4$, $\tau_3 = \tau_4$, and $e''_3 = e''_4$. By transitivity we have $F_1 = F_2$, $e'_1 = e'_2$, $x_1 = x_2$, $\tau_1 = \tau_2$, and $e''_1 = e''_2$.

The remaining case, in which one function appears in p but the other does not, proceeds similarly. The only difference in this case is that, effectively and without loss of generality, $\theta''_4 = []$. □

The remaining clause of Definition 5.6 is the most interesting as it demonstrates the relationship between the (substituted) closure environment and the (substituted) body of any function in a λ_{ITS} program. This lemma makes thorough use of closure consistency as well as previous lemmas, but we use much the same strategy in the previous lemma to merge cases.

Lemma 5.12. If p is closure consistent and $p \longrightarrow p'$ then, for every pair of functions $\lambda^{F_1}_{\ell,e',\theta_1} x_1 : \tau_1 \,.\, e''_1$ and $\lambda^{F_2}_{\ell,e',\theta_2} x_2 : \tau_2 \,.\, e''_2$ appearing in p', we have $\theta_1(e'_1) \simeq \theta_2(e'_2)$ implies $\theta_1(e''_1) \simeq \theta_2(e''_2)$.

Proof. For any function appearing in p', it either appears in p or it does not. We thus have three cases: either both functions appear in p, only one function appears in p, or neither function appears in p. In the first case, we are immediately finished because p is closure consistent.

Consider the case in which neither function appears in p. By Lemma 5.11, we have $F_1 = F_2$, $e'_1 = e'_2$, $x_1 = x_2$, $\tau_1 = \tau_2$, and $e''_1 = e''_2$. For clarity, let $F = F_1$, $e' = e'_1$, $x = x_1$, $\tau = \tau_1$, and $e'' = e''_1$; then we are considering two functions which appear in p' but not in p which are written $\lambda^{F}_{\ell,e',\theta_1} x : \tau \,.\, e''$ and $\lambda^{F}_{\ell,e',\theta_2} x : \tau \,.\, e''$. It remains to show that, if $\theta_1(e') \simeq \theta_2(e')$, then $\theta_1(e'') \simeq \theta_2(e'')$.

We observe by inspection of the rules in Figures 16 and 17 that any function appearing in p' which does not appear in p is the result of substitution of a function appearing in p. There must then exist a function e_3 in p such that $\theta'_3(e_3) = \lambda^{F}_{\ell,e',\theta_1} x : \tau \,.\, e''$ for some θ'_3. Similarly, there must exist a function e_4 in p such that $\theta'_4(e_4) = \lambda^{F}_{\ell,e',\theta_2} x : \tau \,.\, e''$ for some θ'_4.

By assumption, we have $\theta_1(e') \simeq \theta_2(e')$. Let $X' = \text{FV}(e') \cup \text{FTV}(e')$; then, by the argument of Lemma 4.7, we have $(\theta_1|_{X'}) \simeq (\theta_2|_{X'})$. By Lemma 5.10 and because p is closure consistent, we have $x \notin \text{DOM}(\theta_1)$ and $x \notin \text{DOM}(\theta_2)$; thus, $(\theta_1|_{X'}) \simeq (\theta_1|_{X' \cup \{x\}})$ and $(\theta_2|_{X'}) \simeq (\theta_2|_{X' \cup \{x\}})$.

Let $X'' = \text{FV}(e'') \cup \text{FTV}(e'')$. Because p is closure consistent and e_3 appears within p, e' is a canonical closure of x, τ, and e''. By Definition 5.5, $X' = X'' \backslash \{x\}$. We have two cases: either $X'' = X'$ or $X'' = X' \cup \{x\}$. In either case, the above properties of substitutions give us that $(\theta_1|_{X''}) \simeq (\theta_2|_{X''})$. By Definition 4.4, we have $(\theta_1|_{X''}(e'')) \simeq (\theta_2|_{X''}(e''))$. By the argument of Lemma 4.6 and because X'' contains the free variables of e'', we have $\theta_1(e'') \simeq \theta_2(e'')$ and we are finished. □

We now combine the previous four lemmas to formalize closure consistency preservation.

Lemma 5.13. If p is closure consistent and $p \longrightarrow p'$ then p' is closure consistent.

Proof. By Definition 5.6 and Lemmas 5.9, 5.10, 5.11, and 5.12. □

Finally, by combining this most recent result with the base case for initial programs above, we prove that initial programs demonstrate closure consistency throughout execution.

Theorem 1. Let p_0 be an initial program such that $p_0 \longrightarrow^* p_n$. Then p_n is closure consistent.

Proof. By induction on the length n of the evaluation chain, proving the base case with Lemma 5.8 and the inductive step with Lemma 5.13. □

Theorem 1 serves as the basis by which we justify our conservative model of function equality, but it is not itself sufficient to do so. This theorem shows that *syntactically* identical program points and equivalent environments imply *syntactically* equivalent function bodies. To make practical use of intensional function equality, however, a programmer needs to be able to evaluate whether two closures are equal during the program's execution and thus relies upon e.g. instances of the Eq typeclass to determine if two closures are equal. Of course, a faulty implementation of equality could easily produce undesirable results.

For intensional function equality to conservatively approximate *semantic* function equality, we insist upon one additional intuitive requirement: that, for any two values captured in the closures of intensional functions, semantic equality via Eq implies operational equivalence. Since two syntactically equivalent λ_{ITS} terms in the same evaluation context are operationally equivalent, we can use the conclusions of Theroem 1 to support a broader argument that operationally equivalent program points and environments imply operationally equivalent functions.

As stated above, the practical expectation is that the programmer has written in a higher-level language which encodes exclusively into initial λ_{ITS} programs. Our GHC extension follows this process in spirit, encoding intensional functions into a form that guarantees closure consistency. As a result, the programmer may assume the conclusions drawn from Theorem 1.

5.4 Type Checking

We now present a type system for λ_{ITS}. As with the operational semantics, the type system is driven not by a specific novel feature but by the combination and specialization of existing type theory. In particular, we include notions of qualified types [Jones 1992], existential types [Mitchell and Plotkin 1988], and Leibniz equality [Baars and Swierstra 2002; Cheney and Hinze 2002a; Sheard 2005; Weirich 2000; Yakeley 2008].

As mentioned in Section 5.1, λ_{ITS} must support branch-aware runtime type checking. The expression `tyrep` τ_1 `~` `tyrep` τ_2 `?` e_3 `:` e_4 reduces to e_3 if $\tau_1 = \tau_2$ and e_4 otherwise. Reduction to e_3 should also provide that $\tau_1 = \tau_2$ so that e.g. a function expecting τ_1 may accept τ_2 in that branch. We support this behavior via a standard most general unifier (MGU) relation [Pierce 2002; Robinson 1965], defined as follows to acommodate our list-based representation of substitutions.

Definition 5.14. We write $\tau_1 \overset{\theta}{\sim} \tau_2$ to denote that θ is a most general unifier of τ_1 and τ_2. Specifically, let substitution equivalence $\simeq$ be defined for λ_{ITS} in a fashion similar to Definition 4.4. Then $\tau_1 \overset{\theta}{\sim} \tau_2$ iff (1) $\theta(\tau_1) = \theta(\tau_2)$; and (2) for any other θ'_1 such that $\theta'_1(\tau_1) = \theta'_1(\tau_2)$, there exists some θ'_2 and θ'' such that $\theta'_1 \simeq \theta'_2$ and $\theta'_2 = \theta \parallel \theta''$.

The typing relation of λ_{ITS} uses two mappings, C and Γ, from Figure 13. We overload square brackets to denote the lookup of an item by key (e.g. $\Gamma[x]$) and the insertion of a key-value pair (e.g. $\Gamma[x \mapsto \sigma]$). We define the typing of expressions in λ_{ITS} as follows:

Definition 5.15. Let $C; Q; \Gamma \vdash e : \sigma$ be the least relation satisfying the rules in Figure 18.

In addition to a typical type environment, expression, and checked type, this relation includes places for a constraint name mapping C and a constraint set Q. C tracks the type of each constraint function, allowing us to determine e.g. the type of Eq when its corresponding constraint is mentioned. Q indicates available constraint instances and mirrors W of the operational semantics. These structures are fundamental to the T-Witness rule, which consults Q to ensure that the constraint is satisfied and consults C to determine the type of its implementation.

$$\textsc{T-Clo}\ \frac{C; Q; \Gamma \vdash e : \tau \qquad F\ \tau \in Q}{C; Q; \Gamma \vdash \langle F, e, \tau' \rangle : \texttt{clo}\ F} \qquad \textsc{T-TRep}\ \frac{}{C; Q; \Gamma \vdash \texttt{tyrep}\ \tau : \texttt{tyrep}\ \tau}$$

$$\textsc{T-Lam}\ \frac{C; Q; \Gamma \vdash \theta(e') : [\texttt{clo}\ F] \qquad C; Q; \Gamma[x \mapsto \theta(\tau)] \vdash \theta(e) : \tau'}{C; Q; \Gamma \vdash (\lambda^{F}_{\ell, e', \theta} x{:}\tau\,.\,e) : \theta(\tau) \xrightarrow{F} \tau'}$$

$$\textsc{T-TLam}\ \frac{C; Q \cup Q'; \Gamma \vdash e : \sigma \qquad \alpha \notin \textsc{ftv}(Q, \Gamma)}{C; Q; \Gamma \vdash (\Lambda\alpha.\, Q' \Rightarrow e) : (\forall \alpha.\, Q' \Rightarrow \sigma)} \qquad \textsc{T-App}\ \frac{C; Q; \Gamma \vdash e_1 : \tau \xrightarrow{F} \tau' \qquad C; Q; \Gamma \vdash e_2 : \tau}{C; Q; \Gamma \vdash e_1\ e_2 : \tau'}$$

$$\textsc{T-TApp}\ \frac{C; Q; \Gamma \vdash e : \forall \alpha.\, Q' \Rightarrow \sigma \qquad [\alpha \mapsto \tau](Q') \subseteq Q}{C; Q; \Gamma \vdash e\ \tau : [\alpha \mapsto \tau](\sigma)} \qquad \textsc{T-Witness}\ \frac{F\ \tau \in Q \qquad C[F] = \forall \alpha'.\, \tau'}{C; Q; \Gamma \vdash F\ \tau : [\alpha' \mapsto \tau](\tau')}$$

$$\textsc{T-Ident}\ \frac{C; Q; \Gamma \vdash e : \tau \xrightarrow{F} \tau'}{C; Q; \Gamma \vdash \texttt{identify}\ e : \texttt{ppt}} \qquad \textsc{T-Inspect}\ \frac{C; Q; \Gamma \vdash e : \tau \xrightarrow{F} \tau'}{C; Q; \Gamma \vdash \texttt{inspect}\ e : [\texttt{clo}\ F]}$$

$$\textsc{T-Pack}\ \frac{C; Q; \Gamma \vdash e : \tau \qquad F\ \tau \in Q}{C; Q; \Gamma \vdash \texttt{pack}\ e\ \texttt{as}\ F\ \tau : \texttt{clo}\ F}$$

$$\textsc{T-Unpack}\ \frac{\begin{array}{c} C; Q; \Gamma \vdash e : \texttt{clo}\ F \\ C; Q \cup \{F\ \alpha\}; \Gamma[x_1 \mapsto \alpha][x_2 \mapsto (\texttt{tyrep}\ \alpha)] \vdash e' : \sigma \qquad \alpha \notin \textsc{ftv}(Q, \Gamma, \sigma) \end{array}}{C; Q; \Gamma \vdash \texttt{unpack}\ x_1 : \exists \alpha\ \texttt{as}\ x_2 = e\ \texttt{in}\ e' : \sigma}$$

$$\textsc{T-Let}\ \frac{C; Q; \Gamma \vdash e : \sigma \qquad C; Q; \Gamma[x \mapsto \sigma] \vdash e' : \sigma'}{C; Q; \Gamma \vdash \texttt{let}\ x{:}\sigma = e\ \texttt{in}\ e' : \sigma'}$$

$$\textsc{T-Like}\ \frac{\begin{array}{c} C; Q; \Gamma \vdash e_1 : \texttt{tyrep}\ \tau_1 \\ C; Q; \Gamma \vdash e_2 : \texttt{tyrep}\ \tau_2 \qquad \tau_1 \overset{\theta}{\sim} \tau_2 \qquad C; \theta(Q); \theta(\Gamma) \vdash e_3 : \sigma \qquad C; Q; \Gamma \vdash e_4 : \sigma \end{array}}{C; Q; \Gamma \vdash e_1 \sim e_2\ ?\ e_3\ :\ e_4 : \sigma}$$

(omitted for brevity: rules for x, ℓ, `true`, `false`, `==`, $\texttt{nil}^{\tau}$, `::`, `nil?`, and `if`)

Fig. 18. λ_{ITS} Expression Type Checking

Most of the rules presented in Figure 18 are typical for a type system supporting these features. We discuss here the rules which are most relevant to typing intensional functions and their applications. The T-Lam rule is unusual, for instance, in that it performs substitution operations on the expressions that it is typechecking. While substitutions generated by the *evaluation* of expressions would call the decidability of the type system into question, the substitutions performed here are already part of the expression being typechecked and do not present such a difficulty. To establish decidability, we consider a function which eagerly performs all of the substitutions suspended in intensional functions throughout the program. The result of that function serves as evidence for a well-founded induction to establish that this type system is otherwise syntax directed. Incidentally, we define this substitution function in this paper's supplemental material as part of our soundness proof.

$$\text{T-Prog}\ \frac{\begin{array}{c} C = \{F \mapsto \forall\alpha.\,\tau \mid (\texttt{class}\ F : \forall\alpha.\,\tau\,;) \in \overline{c}\} \\ W = \left\{q \mapsto e' \;\middle|\; (\texttt{instance}\ q = e'\,;) \in \overline{d}\right\} \qquad Q = \{q \mid (q \mapsto e') \in W\} \\ \forall(F\ \tau \mapsto e') \in W.\, \exists(F \mapsto \forall\alpha.\,\tau') \in C.\, C; Q; \emptyset \vdash e' : [\alpha \mapsto \tau](\tau') \qquad C; Q; \emptyset \vdash e : \sigma \end{array}}{\vdash \overline{c}\ \overline{d}\ e : \sigma}$$

Fig. 19. λ_{ITS} Program Type Checking

The T-Lam rule is also notable because it requires e' to have type `[clo F]` where F is the constraint function of the intensional function being typed. `clo` F is a bounded existential type satisfying F. This ensures that the closure of the intensional function is the list of type-tagged values expected in Section 5.3. These values can be extracted with the `inspect` projector via the T-Inspect rule.

The type-tagged values in these closures are packed using the T-Pack rule, which matches typical presentations of bounded existential types. The T-Unpack rule is somewhat unusual in that the `unpack` expression syntax includes *three* bindings: one for the packed value, a second for the type of that value, and a third for a runtime witness of that type (in the form `tyrep` τ). This type witness is used in runtime type comparison expressions $e_1 \sim e_2\ ?\ e_3 : e_4$ similar to (but without the elaborate GADT machinery of) `Refl` in Haskell [Baars and Swierstra 2002; Yakeley 2008].

Having defined expression typechecking, we now define program typechecking:

Definition 5.16. Let $p : \sigma$ be the least relation satisfying the rules in Figure 19.

This definition consists of a single rule which, like E-Prog, creates appropriate environmental structures from the program and then handles the program's body expression. This rule also checks to ensure that each typeclass instance's expression conforms to the type given in its typeclass.

We assert the soundness of λ_{ITS} as follows:

Theorem 2 (Soundness)**.** Suppose $\vdash p : \sigma$. Then either p is of form $\overline{c}\ \overline{d}\ v$ or there exists some p' such that $p \longrightarrow p'$ and $\vdash p' : \sigma$.

The proof of this theorem proceeds first by encoding λ_{ITS} in another language established to be sound and then proving that the properties of the encoding together with the soundness of the target language imply the soundness of λ_{ITS}. The encoding process is generally unremarkable in light of previous work: λ_{ITS} is a form of System F [Girard 1971; Reynolds 1974] extended with existential types [Mitchell and Plotkin 1988], runtime type witnesses [Baars and Swierstra 2002; Cheney and Hinze 2002a; Sheard 2005; Weirich 2000; Yakeley 2008], degenerate type classes [Hall et al. 1996], and a typical qualification of types using constraints [Jones 1992]. Each of these features have established encodings [Cheney and Hinze 2002b; Hall et al. 1996; Pottier and Gauthier 2006b; Xi et al. 2003] into System F extended with GADTs, which has been proven sound in many forms (e.g. [Sulzmann et al. 2007; Xi et al. 2003] among others). Our proof appears in this paper's supplemental material.

5.5 Discussion By Example

We now illustrate this system by discussing an example program we present incrementally. We begin with a small code fragment which defines the notion of equality and specifies the behavior of equality on program points.

```
1 class Eq: ∀a. a ⟶[Eq] a ⟶[Eq] bool;
2 instance Eq ppt = λ[Eq]1,[],[] x:ppt. λ[Eq]2,[pack x as Eq ppt],[] y:ppt. x == y;
```

The constraint function `Eq` is associated with a typical type signature for equality. The instance defines equality for `ppt`, the type of program points, in terms of a primitive built for this purpose. Note that the inner function captures the value `x`, of type `ppt`, in closure. This incurs an interesting typing burden: we must prove that `ppt` is `Eq`, and this is what we are in the process of defining! This burden inspired our choice to make top-level instances in λ_{ITS} recursively bound. This choice is not out of place: Haskell, for instance, has the same need for recursive instance bindings to support instances on recursive data types.

We next define equality on booleans:

```
instance Eq bool = λ^Eq_3,[],[] x:bool. λ^Eq_4,[pack x as Eq bool],[] y:bool.
    if x then y else if y then false else true;
```

This definition follows the same structure as with program points above. Our next step is to define equality on individual closure items which indicate that they are equatable:

```
instance Eq (clo Eq) = λ^Eq_5,[],[] x:clo Eq. λ^Eq_6,[pack x as Eq (clo Eq)],[] y:clo Eq.
    unpack vx:∃ tx as rx = x in
    unpack vy:∃ ty as ry = y in
    rx ~ ry ? (Eq tx) vx vy : false ;
```

This example illustrates the need for runtime type comparison: although we know that the elements in the two closures have definitions of equality, we must know that they are the same type to apply such a definition to both elements. We accomplish this by unpacking both existentials to obtain their values and their runtime type representatives. If these representatives match, then we know the types are the same and can use the equality definition of either value to compare them both. Otherwise, the values are not of the same type and so we know they are not equal.

We next define equality on lists of closures:

```
instance Eq [clo Eq] = λ^Eq_7,[],[] x:[clo Eq]. λ^Eq_8,[pack x as Eq ([clo Eq])],[] y:[clo Eq].
        if nil? x and nil? y then true else if nil? x or nil? y then false else
        (Eq (clo Eq)) (hd x) (hd y) and (Eq [clo Eq]) (tl x) (tl y) ;
```

This definition simply compares the lists' elements pointwise, relying upon the definition of closure item equality above. We can finally give a definition for equality between two intensional functions:

```
instance Eq (bool -Eq-> bool) =
    λ^Eq_9,[],[] x:bool -Eq-> bool. λ^Eq_10,[pack x as Eq (bool -Eq-> bool)],[] y:bool -Eq-> bool.
        (Eq ppt) (identify x) (identify y) and (Eq [clo Eq]) (inspect x) (inspect y) ;
```

This definition compares the identities and closures of the intensional functions as discussed in Section 2 (recall Figure 5). While the simplified typeclasses of λ_{ITS} do not permit polymorphism in the function's domain or codomain, our Haskell+`ItsFn` implementation does.

We can finally create two functions using different expressions and compare the functions themselves for equality:

```
let f:(bool -Eq-> bool) = (Eq bool) true in
let g:(bool -Eq-> bool) = (Eq bool) true in
(Eq (bool -Eq-> bool)) f g
```

Our two functions are partially-applied boolean equality functions, both of which have captured `true` in closure and both of which have the program label 4. As a result, they are considered equal and this program evaluates to `true`.

6 Implementation

6.1 Intensional Functions

We have implemented[Palmer and Filardo 2024b] the `IntensionalFunctions` extension in a branch of GHC 9.2. Ideally, intensional functions would be well-integrated into the language runtime; intensional functions could, for instance, be given a dedicated heap representation similar to extensional functions to minimize runtime overhead. But this approach is rife with subtle challenges worthy of their own study. What impacts do compiler optimizations such as inlining have on the semantics of intensional functions? Is a compiler permitted to inline a value which would otherwise be captured in closure? Is a compiler permitted to deduplicate identical function definitions within a module or across modules? These and other problems appear surmountable but require careful consideration which we leave to future work.

Our `IntensionalFunctions` extension of GHC is a proof-of-concept which performs a high-level binding-aware encoding. Intensional functions are defined internally using types similar to those in Figure 20. `ItsFun` carries the values produced by the three intensional function eliminators: identification, inspection, and application.

```
1 data ClosureItem c where
2   ClosureItem :: forall c a.
3     (c a, Typeable a) => a -> ClosureItem c
4 data ItsFun c i o =
5   ItsFun Label [ClosureItem c] (i -> o)
```

Fig. 20. Simplified Intensional Functions Encoding

The `ClosureItem` GADT represents values captured in closure while the `Label` type uniquely identifies the definition site of the function. (The actual types used in our encoding are somewhat more elaborate for reasons described below in Section 6.2.) An expression like `\%Eq x -> x + y` is translated internally to `ItsFun lbl [ClosureItem @Eq y] (\x -> x + y)`. Here, `@Eq` is a type application: the `ClosureItem` constructor requires an `Eq` instance for the type of `y`.

This encoding is not merely syntactic. Note that there are *two* free variables in the expression `\x -> x + y`, but we did not capture the operator `(+)` in a `ClosureItem`. In this example, we presume `y` to be *locally* defined while `(+)` is defined at top level in another module. GHC provides top-level bindings by linking and only captures local values in closure at runtime. Our encoding must match this behavior and so performs desugaring after, and with regards to, name resolution.

6.2 Saturated Application

The code in Figure 20 is simplified for presentation. Our Haskell+`ItsFn` implementation uses more elaborate types, which we initially motivate using the λ_{ITS} expression $(\lambda^{\texttt{Eq}}_{1,[],[]}\texttt{x:ppt.}\ \lambda^{\texttt{Eq}}_{2,[\texttt{pack x as Eq}],[]}\texttt{y:ppt.x == y})\ \texttt{a}\ \texttt{b}$.

The variable `x` is captured in the closure of the inner function because it is free where that function is defined. As `x` (of type `ppt` is captured in closure, we must show `Eq ppt`. But note that the overall expression applies both arguments `a` and `b` simultaneously; there is no opportunity for the constraint `Eq ppt` to be used. An uncurried form of this function, when called, would have no need for a proof of this constraint.

The problem is more compelling when considering `itsBind` as described in Section 3, which has type `m a ->%c (a ->%c m b) ->%c m b`. Just as above, calling `itsBind` would require us to prove `c (m a)`. While this may sometimes be satisfiable, it would be onerous to expect of every intensional monad. `itsBind` is typically called with both arguments at once, meaning that the closure for which we are proving this constraint will often be unused.

Table 1. Comparing Plume Implementations

Implementation	Code Lines in Closure Definition				Unit Test Time
	Type Decl.	Type Annot.	Term Defn.	Total	
Extensional	89	106	438	633	1.72
Intensional	10	84	402	496	4.95

Briefly, consider modifying the grammar of λ_{ITS} in three ways. First, we now write functions as $\lambda^{F}_{\ell,e',\theta}\overline{x:\tau}.e$ to allow multiple parameters. Second, we write applications as $e\ \overline{e}$ so function applications may have multiple arguments. Finally, we write function types as $\overline{\tau} \xrightarrow{F} \tau'$ to reflect these changes. Call sites which saturate the callee need not create a closure and therefore do not impose a constraint on the provided arguments. Note that this moves the burden of proving constraints to function calls rather than function definitions, as we do not know until application whether the constraints will be necessary. Our implementation of Haskell+`ItsFn` uses this saturation awareness to ease typing burdens, especially in the case of intensional monads.

6.3 Intensional Plume

As a preliminary test of the usability of Haskell+`ItsFn`, we have reimplemented a program analysis, Plume [Fachinetti et al. 2020], in Haskell *twice*. Plume encodes program behavior as reachability properties in a specialized pushdown automaton, the closure of which is well-suited to an intensional monad as described in Section 3. Our first artifact implements Plume using intensional monads for indexing and lookup. Our second artifact uses extensional functions and manual defunctionalization.

Table ?? shows the number of lines of code used in each implementation to define Plume's deductive closure algorithm. For illustration, we break these line counts into three categories: type declarations (e.g. `data`, `newtype`), type annotations (e.g. function signatures, `instance`), and term definitions (e.g. function bodies). Other code (e.g. comments, `import` declarations) were not included.

We observe that the implementations' term definitions are comparably verbose. Both implementations include type annotation boilerplate; for each closure rule, the intensional implementation repeats a type signature while the extensional implementation declares a typeclass instance. However, the extensional implementation must declare data types to represent defunctionalized continuations (and their explicit closures) as in line 1 of Figure 4a; this alone accounts for more than half of the 25% increase in line count between the implementations. Subjectively, the intensional implementation is much more readable; we elaborate on this difference in the supplemental material associated with this paper.

As mentioned in Section 6.1, Haskell+`ItsFn` is a proof-of-concept extension despite being built on a production-grade compiler. Table ?? includes a rudimentary benchmark: the average of ten single-threaded executions of the Plume unit tests (drawn from the original artifact). We observe that the intensional implementation takes approximately three times longer to produce the same results.

While we have not directly examined Haskell+`ItsFn` to identify the cause of this poor performance, our proof-of-concept implementation has several qualities that help to explain it. First: the GADT encoding in Figure 20 stores its own copy of the function's environment separate from that kept by the GHC runtime, leading to needless allocation and copying. Second: this copy is stored in the form of a linked list, meaning that e.g. `Ord` between two intensional functions could involve numerous indirections. Third: as `ClosureItems` individually capture their constraints, GHC is unable to fuse their implementations of e.g. `Ord` as it would when deriving `Ord` for a constructor of multiple arguments as in a manually defunctionalized artifact.

We suspect that the above problems would be addressed by integrating intensional functions properly with the GHC runtime, giving them a heap representaton extending that of extensional

functions. Intensional functions would then store a single, unboxed environment and a single pointer to the fused implementation of its constraint. This would additionally allow existing optimizations (e.g. tail call optimization) to be applied to intensional functions almost transparently. Although full integration does raise some interesting questions, such as when a compiler can inline or eliminate values from a closure in light of intensional constraints, this proper integration is primarily an engineering task distinct from the theoretical development above and so beyond the scope of this paper.

7 Related Work

Defunctionalization was originally developed [Reynolds 1972] as a whole-program transformation of an untyped program to use a single global dispatch function in place of every higher-order function call. Many related advancements are summarized in the later republication of that work [Reynolds 1998]. Defunctionalization has been extended to simply-typed [Bell et al. 1997; Tolmach and Oliva 1998] and polymorphically-typed [Pottier and Gauthier 2004, 2006a] languages, the latter of which relied upon a GADT-based encoding of function symbols as in our extensional encoding in Section 2. Similarly, modular (as opposed to whole-program) defunctionalization has been developed [Fourtounis et al. 2014] in a fashion similar to our implementation's approach.

Reynolds's original defunctionalization confounds program analyses and optimizers as, on simplistic inspection, all higher-order call sites reach a function containing the bodies of all higher-order functions. Flow analysis has been used to refine generated dispatch functions to make the functions available at specific call sites more apparent [Cejtin et al. 2000]. Recent work [Contractor and Fluet 2020] has combined this technique with polymorphic type support. Defunctionalization has also been used to make higher-order programs more comprehensible to first-order program analysis and transformations [Avanzini et al. 2015; Mitchell and Runciman 2009].

Defunctionalization has been used as a conceptual bridge between first-order and higher-order languages [Danvy and Nielsen 2001]. This work relates, for instance, evaluation contexts as presented in Figure 15 to continuations. Later work [Danvy and Millikin 2009] considers *refunctionalization*, an inverse of defunctionalization, to form similar theoretical connections.

Defunctionalization has traditionally been applied to compilation pipelines [Cejtin et al. 2000; Hutton and Bahr 2016; Palmer and Raty 2018; Tolmach and Oliva 1998] and in similar back-end settings. Interestingly, Reynolds originally conceptualized defunctionalization as a programmer-facing design technique [Reynolds 1998], a perspective which has gained recent traction [Epstein et al. 2011; Koppel 2019; Miller et al. 2014] and which we share here.

One of the most developed programmer-facing uses of defunctionalization is that of serializing function values. CloudHaskell [Epstein et al. 2011] as implemented by the `distributed-closure` library [Tweag I/O Limited 2020] uses GHC-supported static pointers to serve as serializable defunctionalized symbols for *static* functions (which have empty closures). Spores [Miller et al. 2014] in the Scala language perform a similar task but have no static restriction; instead, values captured in closure by spores are required to be instances of `Serializable`. Language support for closure equality was proposed [Appel 1996] long before these works on serialization and closure equality has been applied heuristically in works like the LMS metaprogramming framework [Rompf and Odersky 2010]. Intensional functions generalize these approaches by abstracting over the operation being applied to the closure in question. Although intensional functions are largely compatible with existing approaches to function serialization — we might, for instance, require the underlying function to be `static` when the constraint function is `Serializable` to support a CloudHaskell-like approach — we leave to future work the exploration of whether a more general mechanism exists for supporting constraint functions with specialized constructors such as deserialization.

Section 6.2 discusses saturation-aware application of intensional functions. Optimizations based upon call arity have been thoroughly studied. GHC itself features call arity transformations to aid in optimization [Breitner 2015a] which have been proven sound [Breitner 2015b].

The λ_θ- and λ_{ITS}-calculi in our formalization are lazy substitution calculi: substitution is delayed until function application. Explicit substitution calculi [Abadi et al. 1990] are similar, but represent substitution as an expression form with its own operational semantics. Calculi with explicit representation of substitution have been proposed as a common functional compilation target [Hardin et al. 1996]. We similarly seek to understand semantics after code transformation, but our lazy substitution is more restrictive in a fashion crucial to our proofs: substitutions may only appear suspended in lambda terms rather than in any subexpression.

In Section 5.3, we observe that the lazy substitutions of intensional functions are similar to environments captured in closure: non-locals are associated with substitutions rather than values. Contextual types [Jang et al. 2021; Nanevski et al. 2008] similarly differentiate between locals and non-locals using an explicit `box` construction. Similarly, modal logic has been used to represent functions capturing non-local values as decomposable structures [Licata et al. 2008] with the goal of unifying binding and computation under a single logical framework.

8 Conclusions and Future Work

We have presented *intensional functions*, a type of function which supports user-defined operations other than application. Such operations are defined in terms of two new eliminators: identification, which yields a unique identity for a given function (in terms of its definition site in the program); and inspection, which produces the values the function has captured in closure. Intensional functions impose type constraints such as equality or orderability on their closures which may then be used to define e.g. conservative equality on the functions themselves.

We have formalized a lazy substitution lambda calculus, λ_{ITS}, supporting intensional functions and defined operational semantics and a type system for it. We have also proven the correctness of conservative equality on intensional functions. This proof can be used as the basis for other correctness arguments using various type constraints.

Our `IntensionalFunctions` GHC extension is not merely syntactic: it uses the scope of bindings to eliminate unnecessary closure elements and provides a form of saturation-aware application to avoid constructing unused closures. We have demonstrated its robustness by reimplementing a program analysis using an intensional coroutine monad. Our extension is, however, a proof of concept; a more complete implementation would integrate with the language runtime to reduce overhead and benefit from optimization decisions made later in the compiler pipeline. We leave these engineering tasks and the theoretical questions they inspire to future work.

Data Availability Statement

The sources for the proof-of-concept Haskell+`ItsFn` compiler can be found on GitHub.com[Palmer and Filardo 2024b]. This compiler is accompanied by a standard library for intensional functions[Palmer and Filardo 2024c], a library defining modular deductive closure engines[Palmer and Filardo 2024a], and implementations of the Plume program analysis with[Palmer and Filardo 2024d] and without[Palmer et al. 2024] intensional functions. A pre-built virtual machine image is available on Zenodo[Palmer 2024].

References

M. Abadi, P. L. Curien, and J. J. Levy. 1990. Explicit substitutions. In *POPL 1990*. ACM Press, San Francisco, California, United States. https://doi.org/10.1145/96709.96712

Andrew W. Appel. 1996. Intensional equality ;=) for continuations. *ACM SIGPLAN Notices* 31, 2 (Feb. 1996), 55–57. https://doi.org/10.1145/226060.226069

Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. Higher-Order Complexity Analysis: Harnessing First-Order Tools.

Arthur I. Baars and S. Doaitse Swierstra. 2002. Typing dynamic typing. (2002), 157–166. https://doi.org/10.1145/581478.581494

Jeffrey M. Bell, Françoise Bellegarde, and James Hook. 1997. Type-Driven Defunctionalization. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97)*. Association for Computing Machinery, 25–37.

Max Bolingbroke. 2011. Constraint Kinds for GHC. http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/

Joachim Breitner. 2015a. Call Arity. In *Trends in Functional Programming*. Springer International Publishing, 34–50.

Joachim Breitner. 2015b. Formally proving a compiler transformation safe. *ACM SIGPLAN Notices* 50, 12 (Aug 2015), 35–46.

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-Directed Closure Conversion for Typed Languages. In *Programming Languages and Systems*. Springer Berlin Heidelberg, 56–71.

James Cheney and Ralf Hinze. 2002a. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 90–104.

James Cheney and Ralf Hinze. 2002b. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, Pittsburgh Pennsylvania, 90–104. https://doi.org/10.1145/581690.581698

John Cocke. 1969. *Programming Languages and Their Compilers: Preliminary Notes*. New York University, USA.

Maheen Riaz Contractor and Matthew Fluet. 2020. Type- and Control-Flow Directed Defunctionalization. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL 2020)*. Association for Computing Machinery, 79–92.

Olivier Danvy and Kevin Millikin. 2009. Refunctionalization at work. *Science of Computer Programming* 74 (Jun 2009), 534–549.

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at work *(PPDP '01)*. 162–174.

Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. Association for Computing Machinery, 118–129.

Leandro Fachinetti, Zachary Palmer, Scott F. Smith, Ke Wu, and Ayaka Yorihiro. 2020. A Set-Based Context Model for Program Analysis. In *Programming Languages and Systems*. Springer International Publishing, 3–24.

Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103, 2 (1992), 235–271.

Georgios Fourtounis, Nikolaos Papaspyrou, and Panagiotis Theofilopoulos. 2014. Modular polymorphic defunctionalization. *Computer Science and Information Systems* 11 (2014), 1417–1434.

J. Y. Girard. 1971. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Scandinavian Logic Symposium*. 63–92.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems* 18, 2 (March 1996), 109–138. https://doi.org/10.1145/227699.227700

Thérèse Hardin, Luc Maranget, and Bruno Pagano. 1996. Functional back-ends within the lambda-sigma calculus. In *ICFP '96*. ACM Press. https://doi.org/10.1145/232627.232632

Graham Hutton and Patrick Bahr. 2016. *Cutting Out Continuations*. Springer International Publishing, Cham.

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2021. Moebius: Metaprogramming using Contextual Types – The stage where System F can pattern match on itself (Long Version). arXiv:2111.08099 (Nov. 2021). https://doi.org/10.48550/arXiv.2111.08099 arXiv:2111.08099 [cs].

Mark P. Jones. 1992. A Theory of Qualified Types. *Sci. Comput. Program.* 22 (1992), 231–256.

Tadao Kasami. 1965. *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages*. Technical Report AFCRL-65-758. Air Force Cambridge Research Laboratory. A slightly later edition may be found at https://hdl.handle.net/2142/74304.

James Koppel. 2019. The Best Refactoring You've Never Heard Of. (2019). https://www.pathsensitive.com/2019/07/the-best-refactoring-youve-never-heard.html Compose.

Daniel R. Licata, Noam Zeilberger, and Robert Harper. 2008. Focusing on Binding and Computation. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. IEEE, Pittsburgh, PA, USA, 241–252. https://doi.org/10.1109/LICS.2008.48

Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag, Berlin, Heidelberg, 308–333. https://doi.org/10.1007/978-3-662-44202-9_13

John C. Mitchell and Gordon D. Plotkin. 1988. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems* 3 (Jul 1988), 470–502.

Neil Mitchell and Colin Runciman. 2009. Losing Functions without Gaining Data: Another Look at Defunctionalisation. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. Association for Computing Machinery, 13–24.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic* 9, 3 (June 2008), 1–49. https://doi.org/10.1145/1352582.1352591

Zachary Palmer. 2024. *Intensional Functions Virtual Machine Image.* https://doi.org/10.5281/zenodo.13381352

Zachary Palmer and Nathaniel Wesley Filardo. 2024a. *Intensional Functions closure engine.* https://github.com/zepalmer/intensional-functions-closure-engine

Zachary Palmer and Nathaniel Wesley Filardo. 2024b. *Intensional Functions GHC.* https://github.com/zepalmer/intensional-functions-ghc

Zachary Palmer and Nathaniel Wesley Filardo. 2024c. *Intensional Functions libraries.* https://github.com/zepalmer/intensional-functions-lib

Zachary Palmer and Nathaniel Wesley Filardo. 2024d. *Plume with Intensional Functions.* https://github.com/zepalmer/intensional-functions-plume

Zachary Palmer, Nathaniel Wesley Filardo, and Ke Wu. 2024. *Modular Extensional Plume.* https://github.com/zepalmer/extensional-modular-plume

Zachary Palmer and Charlotte Raty. 2018. A Schematic Pushdown Reachbility Language. DSLDI 2018.

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.

François Pottier and Nadji Gauthier. 2004. Polymorphic Typed Defunctionalization *(POPL '04).* Association for Computing Machinery.

François Pottier and Nadji Gauthier. 2006a. Polymorphic Typed Defunctionalization and Concretization. *Higher Order Symbol. Comput.* 19, 1 (2006), 125–162.

François Pottier and Nadji Gauthier. 2006b. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation* 19, 1 (March 2006), 125–162. https://doi.org/10.1007/s10990-006-8611-7

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, Vol. 2. ACM Press, Boston, Massachusetts, United States.

John C. Reynolds. 1974. Towards a theory of type structure. In *Symposium on Programming.*

John C. Reynolds. 1998. Definitional Interpreters Revisited. *Higher Order Symbol. Comput.* 11, 4 (Dec 1998), 355–361.

J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (Jan. 1965), 23–41. https://doi.org/10.1145/321250.321253

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10).* Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

Itiroo Sakai. 1961. Syntax in universal translation. In *EARLYMT.*

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug 2020).

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018).* Association for Computing Machinery, 783–796.

Tim Sheard. 2005. Putting curry-howard to work. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell (Haskell '05).* Association for Computing Machinery, 74–85.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '07).* ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

Andrew Tolmach and Dino P. Oliva. 1998. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *J. Funct. Program.* 8, 4 (Jul 1998), 367–412.

Tweag I/O Limited. 2020. `distributed-closure`. https://github.com/tweag/distributed-closure.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming* 21 (Sep 2011), 333–412.

Stephanie Weirich. 2000. Type-safe cast: (functional pearl). In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP '00).* Association for Computing Machinery, 58–67.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03).* Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150

Ashley Yakeley. 2008. Witnesses and Open Witnesses. (2008). https://semantic.org/wp-content/uploads/Open-Witnesses.pdf

Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n3. *Information and Control* 10, 2 (1967).

Received 2024-04-02; accepted 2024-08-18