DALI: An Untyped, CBV Functional Language Supporting First-Order Datatypes with Binders

(Summary[*])

Emir Pasalic, Tim Sheard, Walid Taha[†]

ABSTRACT

Writing (meta-)programs that manipulate other (object-) programs poses significant technical problems when the object-language itself has a notion of binders and variable occurrences. Higher-order abstract syntax is a representation of object programs that has recently been the focus of several studies. This paper points out a number of limitations of using higher order syntax in a functional context, and argues that DALI, a language based on a simple and elegant proposal made by Dale Miller ten years ago can provide superior support for manipulating such object-languages. Miller's original proposal, however, did not provide any formal treatment. To fill this gap, we present both a big-step and a reduction semantics for DALI, and summarize the results of our extensive study of the semantics, including the rather involved proof of the soundness of the reduction semantics with respect to the big-step semantics. Because our formal development is carried out for the untyped version of the language, we hope it will serve as a solid basis for investigating type system(s) for DALI.

1. INTRODUCTION

Programs are data. Nothing makes this point stronger than the ever increasing need for reliable programs with verified properties. As software systems become more complex, and play increasingly important roles in critical systems there is an ever increasing need for optimizing, analyzing, verifying and certifying software.

Each one of these tasks involves automatic manipulation of programs or, *meta-programming.* As with any kind of programming, effective meta-programming relies heavily on the presence of the appropriate support from the (meta-) programming language. The goal of this paper is to advocate a novel approach to representing programs in a manner superior to the main contenders available today. Our approach gives rise to a simple equational theory that can be used to reason about the program equivalence of meta-programs.

[*]The complete technical development appears in a technical report available online. This paper focuses on the describing DALI from the point of view of language design and programming.

[†]Sheard and Pasalic are supported by the USAF Air Materiel Command, contract # F19628-96-C-0161, NSF Grants CCR-9803880 and IRI-9625462, and the Department of Defense. Taha is supported by a Postdoctoral Fellowship, funded by the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403.

1.1 Meta-Programming as Programming

It is our thesis that traditional programming language techniques, including those from the operational, categorical, axiomatic, and denotational traditions can be applied equally effectively to meta-programming languages [15]. In many instances, this means that the technical challenge is "internalizing" various meta-level operations, such as quotation [17], evaluation [16; 9; 2; 15; **?**], and type analysis [14; **?**], into a formal programming language, and subjecting them to the same high standards developed by the semantics community. This approach has numerous pragmatic benefits, including:

1. We succeed in magnifying the subtle features of the operations under investigation, and, often times, in addressing them in a systematic and complete manner. From the software engineering point of view, this translates into enhanced safety and reliability.

2. We succeed in assigning a uniform semantics to these operations that must otherwise be carried out in an *ad hoc* fashion. This can be done to the extent that we can provide mathematically verified reasoning principles for these operations in the form of equational theories. From the software engineering point of view, this translates into enhanced correctness.

3. We make these operations available to the programmer in a uniform way, thus providing more him or her with more control over the *behavior* of the system. From the software engineering point of view, this translates into enhanced predictability.

1.2 Synthesis vs. Analysis

There are two different kinds of program manipulation: *program synthesis*, and *program analysis*. The combination of the two is necessary for expressing general *program transformations.* In what follows we outline the state of the art in language support for both synthesis and analysis, and explain how the present work on DALI fits in the context of analysis.

1.2.1 Synthesis and Multi-Stage Programming

Many recent studies have concentrated on language level support for program synthesis: works on multi-level [5; 4; 10; 8] and multi-stage [17; 16; 9; 2; 15; ?] programming languages have investigated basic problems relating to language support needed for program synthesis such as how to build program fragments, how to combine smaller program fragments into larger ones, and how to execute such fragments in a user friendly, hygienic, and type-safe manner. But while multi-stage programming constructs provide good support for the construction and execution of object-code, they provide no support for analysis. In fact, adding constructs for analyzing code fragments can severely weaken the notion of observational equivalence in such languages [?].

1.2.2 Analysis and Higher Order Syntax

In contrast, substantially fewer studies have focused on language level support for program analysis [?; 13; ?]. With few exceptions (see for example Bjorner [?]), the most popular tool for these studies has been *higher order abstract syntax*[12] (HOAS), and have taken place in the context of logic programming languages [?]. In the remainder of this paper we shall (without drawing too fine a distinction) refer to all approaches to syntax that represent object-level binding constructs by meta-language binding constructs in a uniform way as *higher-order abstract syntax*.

A program analysis inspects the *structure* and environment of an object-program and computes some value as a result. Results can be data- or control-flow graphs, or even another object-program with properties based on the properties of the source object-program. Examples of these kinds of meta-systems are: program transformers, optimizers, and partial evaluation systems [7].

Program analyses are particularly difficult to write correctly if they must manipulate terms that have a notion of statically scoped variables. The exact representation of the variable is generally uninteresting, and often requires subtle administrative changes so that it maintains its original "meaning".

The primary example of such administrative changes is α renaming when the "direct" representation of variables is used, and "shift" and "lift" operations when de Bruijn indices are used. The first representation relies, typically, on the use of state, a "gensym" operation, and the second representation is generally considered "too human unfriendly". Because of this, representing object-programs using first order algebraic data structures which use `string`s or other atomic values to represent variables are notoriously hard to manipulate correctly.

A more pressing concern is that implementing such operations once is not enough: They need to be implemented for *each object-language that has binding constructs*. The basic problem is therefore pervasive, it appears in almost every interesting language.

The basic idea that we advocate is to (uniformly) exploit the binding mechanism of the meta-language to implement the binding mechanism(s) of the object-language, i.e. use functions in the meta-language to implement binding in the object-language. At first glance, this looks like a promising idea, but a number of subtle problems arise. We explicate these problems carefully in Section 3. The problems arise because the functions of the meta-language have two properties which, while necessary for their use as functions, get in the way of their use as binding mechanisms. These properties are: extensionality and delayed computation. Extensionality means that one cannot observe the structure of a function, other than by applying it to get a result. Delayed computation means that computations embodied in a function do not occur until the function is applied. What we need is a new kind of binding, without these properties.

In this paper, we develop such a binding mechanism by refining some ideas of Dale Miller's [?]. This new binding mechanism can be incorporated into a functional language with first-order datatypes, and together they can be used to represent variable binding in object-languages. This mechanism can be systematically reused. In addition, we develop a sound syntactic system for reasoning about the equivalence of functional programs that use this new binding mechanism.

1.3 Contribution

The contribution of this paper is simple and focused: a call-by-value operational semantics for an untyped functional programming language with an extension that supports first-order datatypes (FOD) with binders.

We have applied the rigorous standards of language design and semantic analysis to both the host language (the lambda calculus) and the extension and discovered that the two are mutually compatible. The combined language enjoys a non-trivial equational theory where beta convertibility is a congruence, and is therefore unlikely to invalidate known optimizations for a call-by-value functional language.

We believe that our present operationally-based study complements the recent model-theoretic approach of Gabbay and Pitts [?], Hofmann [?], and Fiore, Plotkin, and Turi [?]. For example, whereas Pitts and Gabbay's recent work emphasizes that a type system is required for their language to ensure that "namefulness" doesn't spread everywhere, our language is untyped, and does not appear to give rise to any non-standard "namefulness" problems.

2. HOAS V.S. FIRST ORDER DATATYPES

The precise semantics of (*meta-*)programs depends crucially on the basic properties of the *representation* of object-programs. This question of representation is the focus of the present study.

The essence of the representation we propose goes back at least to Church [?]. The idea is to exploit the binding mechanism of the meta-language to implement the binding mechanism(s) of the object-language. This is also the essence of Pfenning and Elliot [12] and Miller' [?; ?; ?] *higher-order syntax (HOAS)* representation. To illustrate the basic idea of higher-order syntax, consider the definitions of `Term` and `Term'` below.

```
data Term                    data Term'
 = App Term Term               = App' Term' Term'
 | Abs String Term             | Abs' Term' -> Term'
 | Const Int                   | Const' Int
 | Var String
```

In `Term'` we represent the object-language lambda abstraction (`Abs'`) using the meta-language function abstraction. This way, functions such as `id` and `app` are represented by applying the `Abs'` constructor to a meta-language function:

```
-- \ x -> x                  -- \ x -> x
id = Abs "x" (Var "x")       id' = Abs' (\ x -> x)

-- \f -> \ x -> f x          -- \f -> \ x -> f x
app = Abs "f"                app' = Abs' (\ f ->
       (Abs "x"                      Abs' (\ x ->
         (App (Var "f")                  (App' f x)))
              (Var "x")))
```

The HOAS representation (`Term'`) is elegant in that a concrete representation for variables is not needed, and that it is not necessary to invent unique, new names when constructing lambda-expressions which one can only "hope" don't clash with other names.

3. CRITIQUE OF HOAS

This flavor of HOAS seems like a great idea at first, but careful inspection reveals a few anomalies. It works fine for constructing statically known representations, but quickly breaks down when trying to construct or observe a representation in a algorithmic way. We quickly provide a few small examples that illustrate the problems we have encountered.

P_1 **Opaqueness:** We cannot pattern match or observe the structure of the body of an `Abs'`, or any object-level binding, because they are represented as functions in the meta-language, and meta-level functions are extensional.

We can observe this by casting our `Term'` example above into a real program formulated in ML, and noticing that `id` prints as `Abs' fn`.

```
(* Actual ML Program Execution *)

- datatype Term'
  = App' of (Term'* Term')
  | Abs' of (Term' -> Term')
  | Const' of int;

- val id = Abs'(fn x => x);
val id = Abs' fn : Term'
```

P_2 **Junk [?; 3] :** I.e., there are terms in the meta-language with type `Term'` that do not represent any legal object-program. Consider:

```
junk = Abs'(\ x -> case x of App' f y -> y
                           ; Const' n -> x
                           ; Abs' _ -> x)
```

No legal object-program behaves in this way.

P_3 **Latent Divergence:** Because functions delay computation, a non-terminating computation producing a `Term'` may delay non-termination until the `Term'` object is observed. This may be arbitrarily far from its construction, and can make things very hard to debug. Consider the function `bad` below:

```
bad (Const' n) = Const' (n+1)
bad (App' x y) = App' (bad x) (bad y)
bad (Abs' f) = Abs'(\ x -> diverge(bad (f x)))
```

`bad` walks over a `Term'` increasing every explicit constant by one. Suppose the programmer made a mistake and placed an erroneous divergent computation in the `Abs'` clause. Note that `bad` does not immediately diverge.

P_4 **Expressivity:** Using HOAS, there exist (too many) meta-functions over object-terms that cannot be expressed. Consider writing a `show` function for `Term'` that turns a `Term'` into a `string` suitable for printing.

```
show (App' f x) = (show f) ++ " " ++ (show x)
show (Const' n) = toString n
show (Abs' f) = "\\ "++ ?v ++ " -> "
                 ++ (show (f ?v))
```

What legal meta-program value do we use for `?v`? We need some sort of "variable" with type `Term'` but no such thing can be created. There are "tricks" for solving this problem [?], but in the end, they only make matters worse.

Our approach to these problems is to cast our search for solutions as an exercise in programming language design. The following subsections offer an informal discussion of each problem and a potential solution by the introduction of additional language features, and provide examples of how these language features might be used. Our biggest challenge is to discover features that interact well, both with each other, and with the existing features of the language we wish to add them to.

3.1 Opaqueness

To solve the opaqueness problem a number of researchers have investigated the use of higher-order pattern matching [?]. The basic idea is that programmers use a higher-order interface to the object-language because it is expressive and easy to use, but the actual underlying implementation is first order.

One tries to supply an enriched interface that gives programmers access to this first-order implementation in a safe manner, that still supports all the benefits of a higher-order implementation. To illustrate this consider the (not necessarily semantics-preserving) rewrite rule **f** for object-terms `Term'`, which might be expressed as:
$\mathbf{f}: \ (\lambda\, \mathbf{x}.(\mathbf{e}'\, \mathbf{0})) \rightarrow (\mathbf{e}'[\mathbf{0}/\mathbf{x}])$.

Here, we use the notation, that a primed variable is a meta-variable. Thus e' is a meta-variable of the rule, and $e'[0/x]$ indicates the capture free substitution of 0 for x in e'.

Higher-order pattern matching is a programming language mechanism, that allows us to express that we wish to observe the inner structure of meta-language abstraction, and that parts of the body of this abstraction (i.e. e') may have free occurrences of x inside.

We use a higher-order pattern when we wish to analyze the structure of a constructor like `Abs'` which takes a meta-function as an argument. Like all patterns, a higher order pattern "binds" a meta-variable. The meta-variable bound by a higher-order pattern does not bind to an object-term, but instead binds to a function. This function captures the subtlety that e' might have free occurrences of x. Given the bound variable, as input, it reconstructs the body of the abstraction. Given a term as input, it substitutes the term for each free occurrence of the bound variable in the body.

The bound meta-variable is a function from `Term' -> Term'`. We make this language mechanism concrete by extending the notion of pattern in our meta-language. Patterns can now have explicit lambda abstractions, but any pattern-variables inside the body of the lambda abstraction are higher-order pattern-variables, i.e. will bind to functions. Consider below, an example implementing the rewrite rule above.

```
f (Abs'(\ x -> App'(e' x)(Const 0))) = e'(Const 0)
f x = x
```

In this example the meta-function `f` matches its argument against an object-level abstraction (`Abs' ...`) using an object-level pattern. The pattern specifies that the body of the matched abstraction must be an application (`App'`) of a function term (`e' x`) to a constant (`Const 0`). The function part of this object-application can be any term. This term may have free occurrences of the object-bound variable (which we write as `x` in the pattern, but which can have any name in the object-term it matches against). Because of this we use a higher-order pattern (`e' x`) which applies `e` to `x` to indicate that `e'` is a function whose argument is the object-bound variable `x`.

This extension differs from normal pattern matching, in that neither meta-level abstractions (`\ x -> ...`) nor applications of meta-variables `(e' x)` normally appear in regular patterns.

If the underlying implementation is first order (like `Term`), patterns of this form have an efficient and decidable implementation. The clause
`f (Abs'(\ x -> App'(e' x)(Const 0))) = e' (Const 0)`
would translate into an implementation using `Term` as follows:

```
f (Abs x (App e (Const 0))) =
     let e' y = subst [(x,y)] e
     in e' (Const 0)
```

The key advantage of this approach is that users get to use the expressive and safe HOAS interface, and the substitution function need not be written by the programmer but can be supplied by the underlying implementation.

The solution of using (a hidden) underlying first order implementation, but supplying a higher-order interface, extends nicely to term construction as well as term observation.

A construction like: `(Abs' f) :: Term'` could be translated into an underlying implementation based on first-order, observable, data-structures (i.e. `Term`) by using a *gensym* construct to provide a "fresh" name for the required object-bound variable:
`let y = gensym () in Abs y (f (Var y))`.
Again both the *gensym* and the underlying first-order implementation is hidden from the user.

3.2 Junk

Junk is a serious problem in that it allows meta-programs to represent non-existent terms in the object-language. Junk arises because the body of an object-binding is a computation (i.e. a suspended function), rather than a constant piece of data. This causes two kinds of problems:

1. The computation can "observe" the bound variable, and do ill-advised things like pattern matching. A valid object-binding only "builds" new structure around the variable. It does not observe the bound variable.

2. The computation can introduce effects. In this case the computational effects of the meta-language, such as nontermination, are introduced into the purely syntactic representation of the object language. Even worse, the effects are only introduced when the object-term is observed. If a term is observed multiple times, it causes the effects to be introduced multiple times.

3.3 Latent Divergence

So we see that that junk and latent divergence are really two facets of the same problem. To fix these problems we need a binding construct which preserves static scoping (like normal meta-level functions) but which does *not* delay computation. What we need is a binding construct which forces computation "under the lambda" [15].

Ten years ago, Dale Miller proposed a new meta-level binding construct for implementing HOAS in ML [**?**] which did exactly this. He introduced a new binary type constructor (`a => b`) which names the type of an object level binding of `a` terms in `b` terms. The new type constructor was used in place of the function type constructor to denote object-level abstraction.

We introduce DALI, a language based upon a refinement of Miller's idea. We compare it to HOAS, and illustrate its intended use by a number of examples. In DALI, the object-binding mechanism is separate from the function construct of the meta-language. This allows us to restrict the range of junk, and the introduction of erroneous effects. Consider our small lambda calculus example once again.

```
datatype Term
  = App Term Term
  | Abs Term => Term
  | Const Int
```

Terms of type `a => b` are introduced using the meta-language construct for object-binding introduction. The expression level syntax of the meta-language, is analogous to the syntax of the type constructor for object-level bindings. For example: `(#x => App(#x,Const 0)) :: (Term => Term)`. Here we use the hash (`#x`) notation to distinguish object-level variables from meta-level variables. An important property of object-variables, is that they cannot escape their scope. Like meta-level function binding, object-binding respects static scoping. The `=>` introduction construct `(#x => e)` delimits the scope of `#x` to `e`. The key property of object-binding is that evaluation proceeds under =>.

Below are two different examples of constructing an object-language program. The first using meta-level functions as the binding mechanism, and the second using object-level abstraction:

```
  Abs'(\x -> bottom)            Abs(#x => bottom)
```

The expression on the left uses a meta-language binding mechanism (λ abstraction). It succeeds in constructing a representation of an object-language program which obviously has no meaning. The expression on the right, however, does not represent any object-language program, since the expression never terminates. Note that the effect on the left has seeped into the object-language program representation (junk), while on the right non-termination occurs before the object-language program is constructed and thus is never present in the object-language program itself.

A more sophisticated example is the copy function over `Term`

```
copy (App f x) = App (copy f) (copy y)
copy (Const n) = Const n
copy (Abs(#x => e' #x)) = Abs(#y => (copy (e' #y)))
copy (x @ #_) = x
```

To those familiar with functional programming, the first two clauses should be clear. The third clause, uses the higher-order pattern matching introduced earlier, only applied here in the context of the new object-level binding construct. Since evaluation passes under => diverging computations will not delayed.

The fourth clause of the `copy` function is an artifact of the object-level binding mechanism. Object-variables (`#x`) introduced using the object-binding syntax: `(#x => ...)`, are a new type of constant. The actual name of such a constant is not accessible to the programmer. There are two operations that are necessary on object-variables, it should be possible to distinguish them from other object-terms, and it should be possible to compare them using equality, in order to tell them apart.

Thus, functions over object-languages, must have a clause for object-bound variables. Object-bound variables are distinct from all other constructors, and are common to all object-languages. The pattern `#_` matches any object-variable, but fails to match other constructors. The binding says nothing about the name of the variable it binds to. The notation `(x @ #_)` introduces a meta-level variable `x`, bound to the object-level variable matched by the object-pattern `#_`.

3.4 Expressivity

It is sometimes necessary to eliminate object-bound variables. This is done in one of two ways. First by applying a higher-order pattern variable to some value `x :: Term`, the occurrences of the bound variable will be replaced with `x`.

This is not always sufficient since it does not provide any way of transforming a object-binding into anything other than another object-binding. This was the problem with the `show` function (Section 3). This is why HOAS using meta-level function binding cannot express some functions. Object-level binding allows us to solve this problem.

The solution is a new language construct *discharge*. The construct `(discharge #x => e1)` introduces a new object-level variable `(#x)`, whose scope is the body `e1`. The value of the discharge construct is its body `e1`. The body `e1` can have any ground type, unlike an object-level binding `(#x => e2)`, where `e2` must be an object term.

In addition, discharge incurs an obligation that the object variable `(#x)` does not appear in the value of the body `(e1)`. An implementation must raise an error if this occurs.

For example consider a function which counts the number of `Const` subterms in a `Term`.

```
count :: Term -> Int
count (Const _) = 1
count (App f x) = (count f) + (count x)
count (Abs(#x => e' #x)) =
      discharge #y => count (e' #y)
count #_ = 0
```

Note how that the fourth clause conveniently replaces all introduced object-bound variables with 0, thus guaranteeing that no object-variable appears in the result. The obligation that the variable does not escape the body of the discharge construct may require a run-time check (though in this example, since the result has type `Int`, no such occurrence can happen).

If a programmer needs to treat individual object-bound variables in different ways, he can use an environment parameter. Consider the program below, which is the correct implementation of the function `show`.

```
show x = sh n [] x
 where
  sh n (App f x) = (sh n f) ++ " " ++ (sh n x)
  sh n (Const n) = toString n
  sh n (x @ #_) = lookup x n
  sh n (Abs(#y => f #y)) =
     let x = len n
         v = x ++ (toString x)
     in discharge #x =>
        "\\ "++ v ++ " -> "
           ++ (show ((#x,v):n) (f #x))
```

Here the environment `n` is a list of pairs mapping object-variables to strings. If the `sh` function is applied to an object-variable it looks up its name in the environment. For an object-abstraction, `(Abs'(#x => f #x))`, discharge introduces a new object-variable, adds it to the environment, and then applies the higher-order pattern variable `f` to the introduced variable, and recursively produces a string as the representation of the abstraction's body.

Another example transforms a `Term` into its de Bruijn equivalent form.

```
data DB
  = DApp DB DB
  | DAbs DB
  | DVar Int
  | DConst Int

DeBruijn env (App f x) =
  DApp (DeBruijn env f) (DeBruijn env x)
```

```
DeBruijn env (Abs(#x => e' #x)) =
  discharge #y => DAbs(DeBruijn (ext env #y)(e' #y))
    where ext env v u =
              if v=u then 0 else 1 + (env u)
DeBruijn env (Const n) = DConst n
DeBruijn env (z @ #_) = env z
```

4. EXAMPLES

In this section we use our language to express some classic manipulations on object-languages.

- Lambda calculus syntax

```
datatype Lterm = App Lterm Lterm
               | Abs Lterm => Lterm
               | Const Int
               | Prod Lterm Lterm
```

- Call-by-name Big-step evaluator for untyped lambda calculus:

```
eval : Lterm -> Lterm
eval (Abs body) = Abs body
eval (App t1 t2)=
 case eval t1 of
   (Abs (#x => body #x)) -> eval (body t2)
eval (Const n) = Const n
eval (Prod x y) = Prod (eval x) (eval y)
eval (x @ #_) = x
```

- CBN lambda calculus (single step) reduction:

```
beta : Lterm -> Lterm -> Lterm
beta (Abs (#z => body #z)) t2 = body t2
```

- Complete development:

```
compdev : Lterm -> Lterm
compdev (Abs(x# => body #x))
    = Abs(#w => compdev (body #w))
compdev (App (Abs(#x => body #x)) y)
    = sub (Abs(#w => compdev(body #w))) (compdev y)
        where sub (Abs(#z => e #z)) x = e x
compdev (App f x) = App(compdev f)(compdev x)
compdev (Prod x y) = Prod(compdev x)(compdev y)
compdev (Const n) = Const n
compdev (x @ #_) = x
```

- Substitution on Lterms.

```
find x [] = Nothing
find x ((y,v):ys) = if x==y
                    then v
                    else find x ys

subst: Lterm -> [(Lterm,Lterm)] -> Lterm
subst x env =
 case find x env
   Just t -> t
   Nothing ->
     case x of
      v @ #_ -> v
      Abs(#x => e #x) ->
          Abs(#w => subst env (e #w))
      App x y -> App(subst env x)(subst env y)
      Prod x y -> Prod(subst env x)(subst env y)
      Const n -> Const n
```

5. NEW FEATURES OF DALI

The language DALI contains some features that behave in untraditional ways. It is useful to call attention to these features.

- *Object variable bindings:* Unlike the meta-language binding construct, the evaluation of an object-level abstraction (`(#x => e)`) proceeds "under" the `=>`.

- *Ground (or* equality*) values:* such values can be compared for simple structural equality. The important property of ground values is that they do not contain functions. Only ground values are used to represent valid object languages.

  In order to compare object-language terms for equality it is necessary to compare object-variables for equality. This must be a primitive in the language. Equality on object-language types is important for two reasons. First, it facilitates an important programming technique, illustrated in our de Bruijn notation example above. Second, it makes possible higher-order pattern matching (see below).

- *Object-variable matching:* Comparing object-language terms for equality is not enough for the meta-programs in our examples. We must be able to distinguish object-level variables from other object-level terms. This is the purpose of the (`#_`) pattern.

- *Higher-order pattern matching:* A higher order pattern variable (i.e. `x` in `(\(#z => x #z)->e)` is bound to a (meta-level) function that returns the body of the object-level abstraction after replacing the object-variable with its argument. This in effects internalizes substitution for bound object-level variables in object programs. In order to implement such a scheme, it is important that the object-abstraction body be an equality type. I.e. we must somehow disallow types of the form `(a => (b -> c))`. If we do not do this then interesting anomalies may occur. For example consider:

```
f (#z => x #z) = x (Const 0)

w = (#x => (\ y -> Prod #x (Const 5)))
```

  If we apply `f` to `w`, we must build a meta-level function `x` which replaces all occurrences of `#x` in `(\ y -> Prod #x (Const 5))` with `(Const 0)`. It is unlikely we can do this if functions are only extensional.

6. A NOTE ABOUT "DISCHARGE"

In defining meta-programs, the use of `discharge` is often crucial, since it allows for eliminating an object-level binding and performing computations only on its body. However, the binding's body can be safely extracted only if there is a guarantee that a heretofore bound object-level variable cannot become free as a result of computation over its body. There are two ways of adding `discharge` to DALI: First, as a new language construct with appropriate reduction rules; and second, as a function defined by the user on a per-datatype basis.

In the present paper, we opt for the second design decision in order to keep the core calculus of DALI, and its technical development as small as possible. We present an example of a user defined `discharge` function for the lambda-term datatype (`Lterm`):

```
discharge (#w => t #w) =
    case (#a => find #a (t #a)) of
       (#z => True ) -> t ()
       (#z => False) -> diverge

find var (App t1 t2) =
   (find var t1) or (find var t2)
find var (Abs(#w=>b #w)) =
    case (#z => find var (b #z)) of
           (#z => True )  -> True
           (#z => False)  -> False
find var (Prod t1 t2) =
   (find var t1) or (find var t2)
find var (Const n) = False
find var (x @ #_) =
   if x = var then True
              else False
```

The function `discharge` simply searches the body of an object-level abstraction for the abstracted object-level variable. If the variable is not found in the body, the program simply returns the body itself. Otherwise, the computation diverges.

It is important to note that in DALI, `discharge`, whether added as a language construct or defined as a function, can be used only on ground values, i.e., values that do not contain suspended computation. Extending `discharge` to apply to values that contain abstraction causes confluence and soundness problems similar to those described in section 8.1.

However, there appear to be situations where such a more general version `discharge` is desirable. The example below, implements a kind of evaluation for the familiar encoding of untyped lambda terms, using an environment.

```
data Value = Vint Int
           | Vprod Value Value
           | Vfun  Value -> Value

type Env = [(Lterm * Value)]

eval' : Env -> Lterm -> Value
eval' env (Abs (#x => b #x)) =
  discharge #w =>
    Vfun(\ y -> eval' (extend #x y env) (b #w))
eval' env (App t1 t2) =
 case eval' env t1 of
    Vfun f -> f (eval' env t2)
eval' env (Const n) = Vint n
eval' env (Prod x y)
  = Prod(eval' env x)(eval' env y)
eval' env (x @ #_) = env x
```

In the present version of the language it is impossible do define a `discharge` function needed for the second clause of `eval'`, since it would involve detection of free object-bound variables in a term that contains an (extensional) meta-level function. On the other hand, the example is intuitively correct, and one can convincingly argue from the definition of the function `eval'` that the discharged object-level variables indeed never do appear in the values of `eval'`. Whether an appropriate mechanism can be introduced to extend `discharge` to such cases remains a question yet to be fully addressed for DALI.

7. FORMAL SEMANTICS OF CORE DALI

7.1 Syntax

Figure 1 defines the various syntactic categories used in specifying Core DALI, including expressions $\mathbb{E}$, ground values $\mathbb{B}$, values $\mathbb{V}$, and contexts $\mathbb{C}$.

Expressions in Core DALI include the lambda calculus with naturals. Further, the language incorporates datatypes (not necessarily just first-order), in addition to the following specialized mechanisms:

- Object-level variables $\#z$ and binders $(\#z \Rightarrow e)$,
- Pattern matching over object-bindings $\lambda(\#z \Rightarrow x).e$.
- Equality for object-bound variables $\#z =_{\#} \#z'$
- Test of whether an expression evaluates to an object-level variable (isOVar e).

Values, ground values, and context are used in defining the reduction semantics.

7.2 Core DALI vs. Example Language

The Core DALI has two (more primitive) forms of pattern matching than the language used in the examples: one for tagged values, one for object-level bindings. A third form of pattern matching (for object-level variables) can be easily encoded using isOVar.

Nested patterns are not allowed, nor are more complicated higher-order patterns directly supported: each constructor has one argument, and each higher-order pattern variable has exactly one possible free object variable in it. These simplifications make the formal development of Core DALI more manageable, without losing generality: programs in a more familiar language of our examples can be translated into equivalent, albeit more verbose Core DALI expressions.

7.3 Big Step Semantics (λ^D)

Figure 2 defines the call-by-value (CBV) big-step semantics for Core DALI. Note that this semantics does not require a *gensym* function or any freshness conditions on variables: All necessary variable renaming is handled by two standard notions of substitution [1], one for object-level variables ($\#z \in \mathbb{Z}$) and one for meta-level variables ($x \in \mathbb{X}$).

7.4 Reduction Semantics (λ^d)

Figure 1 defines the reduction semantics for Core DALI.

8. SUMMARY OF TECHNICAL DEVELOPMENT

The main technical result of our work to date is establishing the confluence property for the reduction semantics described above, and establishing (the rather non-trivial connection) between the reduction semantics and big-step semantics. In doing so, we have following closely Taha's development for the (substantially smaller) language $\lambda{-}U$ [15; ?].

Syntax:

$x \in \mathbb{X}$	Normal variables	$:=$	Infinite set of names
$z \in \mathbb{Z}$	Object variables	$:=$	Infinite set of names
$f \in \mathbb{F}$	Tags	$:=$	Infinite set of names containing True and False
$F \subset \mathbb{F}$	Tag sets	$=$	Finite subsets of $\mathbb{F}$
$e \in \mathbb{E}$	Expressions	$:=$	$() \mid x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_1\ e \mid \pi_2\ e \mid f\ e \mid \lambda^{f\in F}(f\ x_f).e_f \mid$ $\#z \mid \#z \Rightarrow e \mid \lambda(\#z \Rightarrow x).e \mid \mathsf{isOVar}\ e \mid e =_{\#} e$
$C \in \mathbb{C}$	Contexts	$:=$	$[] \mid \lambda x.C \mid C\ e \mid e\ C \mid (e,C) \mid (C,e) \mid \pi_1\ C \mid \pi_2\ C \mid$ $\lambda^{f\in F-\{f'\}}((f_i\ x_i).e_i) ++ (f'\ x).C \mid f\ C \mid (\#z \Rightarrow C) \mid \lambda(\#z \Rightarrow x).C \mid$ $\mathsf{isOVar}\ C \mid C =_{\#} e \mid e =_{\#} C$
$b \in \mathbb{B}$	Ground Values	$:=$	$() \mid (b,b) \mid f\ b \mid \#z \mid \#z \Rightarrow b$
$v \in \mathbb{V}$	Values	$:=$	$() \mid \lambda x.e \mid (v,v) \mid f\ v \mid \lambda^{f\in F}\ f\ x_f.e_f \mid \#z \mid \#z \Rightarrow v \mid \lambda(\#z \Rightarrow x).e$
$\rho \in \mathbb{R}$	Reductions	$:=$	$\beta_1 \mid \pi_1 \mid \pi_2 \mid \beta_2 \mid \beta_3 \mid \# \mid \delta_{\mathsf{isOVar}}$

Notions of Reduction:

$$
\begin{array}{rcl}
(\lambda x.e)\ v & \longrightarrow_{\beta_1} & e[x := v] \\
\pi_1\ (v_1, v_2) & \longrightarrow_{\pi_1} & v_1 \\
\pi_2\ (v_1, v_2) & \longrightarrow_{\pi_2} & v_2 \\
(\lambda^{f\in F\cup\{k\}}(f\ x_f).e_f)\ (k\ v) & \longrightarrow_{\beta_2} & e_k[x_k := v] \\
(\lambda(\#z' \Rightarrow x).e)\ (\#z \Rightarrow b) & \longrightarrow_{\beta_3} & e[x := \lambda y.(b[\#z := y])] \\
\#z =_{\#} \#z & \longrightarrow_{\#} & \mathsf{True}() \\
\#z_1 =_{\#} \#z_2 & \longrightarrow_{\#} & \mathsf{False}() \text{ if } z_1 \neq z_2 \\
\mathsf{isOVar}\ \#z & \longrightarrow_{\mathsf{isOVar}} & \mathsf{True}() \\
\mathsf{isOVar}\ v & \longrightarrow_{\mathsf{isOVar}} & \mathsf{False}() \text{ if } v \neq \#z
\end{array}
$$

Reduction Semantics:

$$
\frac{e_1 \longrightarrow_{\rho} e_2}{C[e_1] \longrightarrow C[e_2]}\rho \in \mathbb{R} \qquad \frac{}{e \longrightarrow^{*} e} \quad \frac{e_1 \longrightarrow e_2 \quad e_2 \longrightarrow^{*} e_3}{e_1 \longrightarrow^{*} e_3}
$$

Figure 1: Syntax and Reduction Semantics (λ^d) of Core DALI

Taha's development is based on Takahashi parallel reduction and complete development methods for proving confluence [19], and Plotkin's "standardization" technique for showing that reductions preserve observational equivalence.

This section summarizes our technical development and states our main result, and *explains how they were useful to us in the process of designing the semantics for DALI.* The full details cannot be included in this paper, and are presented instead in a technical report available on-line [11, 40 pages].

8.1 Confluence

The first result is confluence:

THEOREM 1 (λ^d IS CONFLUENT). $\forall e_1, e_2, e \in \mathbb{E}.$

$$e_1{}^{*} \longleftarrow e \longrightarrow^{*} e_2 \Longrightarrow (\exists e' \in \mathbb{E}.\ e_1 \longrightarrow^{*} e'^{*} \longleftarrow e_2)$$

First, we are not aware of a similar proof for a language with datatypes. Furthermore, this result establishes the existence of a confluent calculus for a language with notion of object-level binders, *and* analysis on these terms. In particular, this result means that DALI also provides a solution to the problem of introducing intensional analysis to MetaML in a "coherent" manner [**?**].

8.1.1 Role in Design of DALI

In addition to its technical role in arriving at our next result, establishing the confluence property played an important role in our design process: It drew our attention to the need for introducing the notion of ground-values, thereby prohibiting any useful mixing of object-binder and function spaces in datatypes.

In particular, analysis over object-level binders (β_3 reduction) without the restriction of the argument to ground values breaks the confluence, as is illustrated in the following example:

Suppose the notion of reduction $\longrightarrow_{\beta_3}$ (Figure 1) were defined as follows (we emphasize the part different from the standard definition by placing it into a box):

$$(\lambda(\#z' \Rightarrow x).e)\ \boxed{(\#z \Rightarrow v)} \longrightarrow_{\beta_3} e[x := \lambda y.v[\#z := y]]$$

Now, consider the function $f \equiv (\lambda(\#w \Rightarrow x).x\ (\lambda u.u))$. This function takes an object-level binding as its argument and returns the body of the binding in which the object-bound variable has been replaced with the identity function $\lambda u.u$. For the application of f to the object binding $(\#z \Rightarrow (\lambda y.\#z = \#z))$, there are two possible reduction sequences:

$$f\ (\#z \Rightarrow (\lambda y.\#z = \#z)) \quad \longrightarrow_{\beta_3} \lambda y.(\lambda u.u) = (\lambda u.u)$$

and

$$
\begin{array}{rl}
f\ (\#z \Rightarrow (\lambda y.\#z = \#z)) \longrightarrow_{\#} & f\ (\#z \Rightarrow (\lambda y.\mathsf{True}())) \\
\longrightarrow_{\beta_3} & \lambda y.\mathsf{True}()
\end{array}
$$

Clearly, neither $\lambda y.\mathsf{True}()$, nor $\lambda y.(\lambda u.u) = (\lambda u.u)$ can be further reduced by λ^d to a common reduct: a clear counterexample for confluence.

$$\frac{}{() \hookrightarrow ()} \qquad \frac{}{\lambda x.e \hookrightarrow \lambda x.e} \qquad \frac{\begin{array}{l} e_1 \hookrightarrow \lambda x.e \\ e_2 \hookrightarrow e_3 \\ e[x := e_3] \hookrightarrow e_4 \end{array}}{e_1\ e_2 \hookrightarrow e_4} \qquad \frac{\begin{array}{l} e_1 \hookrightarrow \lambda^{f \in F \cup \{k\}}(f\ x_f).e_f \\ e_2 \hookrightarrow k\ e_4 \\ e_k[x := e_4] \hookrightarrow e_5 \end{array}}{e_1\ e_2 \hookrightarrow e_5} \qquad \frac{\begin{array}{l} e_1 \hookrightarrow \lambda(\#z' \Rightarrow x).e \\ e_2 \hookrightarrow \#z \Rightarrow b_3 \\ e[x := \lambda x'.(b_3[\#z := x'])] \hookrightarrow e_4 \end{array}}{e_1\ e_2 \hookrightarrow e_4}$$

$$\frac{e_1 \hookrightarrow e_3 \quad e_2 \hookrightarrow e_4}{(e_1, e_2) \hookrightarrow (e_3, e_4)} \qquad \frac{e \hookrightarrow (e_3, e_4)}{\pi_1\ e \hookrightarrow e_3} \qquad \frac{e \hookrightarrow (e_3, e_4)}{\pi_2\ e \hookrightarrow e_4} \qquad \frac{e_1 \hookrightarrow e_2}{f_k\ e_1 \hookrightarrow f_k\ e_2} \qquad \frac{}{\lambda^{f \in F} f\ x_f.e_f \hookrightarrow \lambda^{i \in F} f\ x_f.e_f}$$

$$\frac{}{\#z \hookrightarrow \#z} \qquad \frac{e_1 \hookrightarrow e_2}{\#z \Rightarrow e_1 \hookrightarrow \#z \Rightarrow e_2} \qquad \frac{}{\lambda(z \Rightarrow x).e \hookrightarrow \lambda(z \Rightarrow x).e} \qquad \frac{\begin{array}{l} e_1 \hookrightarrow \#z \\ e_2 \hookrightarrow \#z \end{array}}{e_1 =_{\#} e_2 \hookrightarrow \mathsf{True}()} \qquad \frac{\begin{array}{l} e_1 \hookrightarrow \#z_1 \\ e_2 \hookrightarrow \#z_2 \quad z_1 \neq z_2 \end{array}}{e_1 =_{\#} e_2 \hookrightarrow \mathsf{False}()}$$

$$\frac{e \hookrightarrow \#z}{\mathsf{isOVar}\ e \hookrightarrow \mathsf{True}()} \qquad \frac{e \hookrightarrow v \quad v \neq \#z}{\mathsf{isOVar}\ e \hookrightarrow \mathsf{False}()}$$

Figure 2: Big-Step Semantics (λ^D) of Core DALI

Finally, note that the breakdown of confluence here provides a concrete illustration of one of the wide range of difficulties that can arise from mixing function spaces with "syntax". Other examples, such as the discussion of "covers" in the context of MetaML implementation [15] require much more infrastructure to present.

8.2 Soundness

We will consider two programs to be equivalent when they can be interchanged in any context without affecting the termination (or non-termination) of the full term in which they occur. This is known as observational (or contextual) equivalence, and is defined as follows:

Definition 2 (Observational Equivalence). *We write $e_1 \approx e_2$ if and only if*

$$\forall C \in \mathbb{C}.\ (\exists v \in \mathbb{V}.\ C[e_1] \hookrightarrow v) \Leftrightarrow (\exists v \in \mathbb{V}.\ C[e_2] \hookrightarrow v)$$

Our soundness result can now be stated as simply:

Theorem 3 (Soundness).

$$\forall e_1, e_2 \in E.\ e_1 \longrightarrow e_2 \Longrightarrow e_1 \approx e_2$$

First, our proof for this theorem is the first operational account known to us where the soundness of such reductions for an untyped CBV functional language with datatypes is established (Using bisimilarity techniques, Pitts does present a similar result, but for a typed CBN language supporting binary sum types [**?**].)

Second, the soundness of these results establishes that extending the lambda calculus plus datatypes with DALI's constructs for introducing and analyzing object-level binders and free variables at runtime *does not* injure the notion of observational equivalence in a devastating way. Certainly, it may very well be that introducing the new constructs allows us to distinguish between more terms in the language (as does introducing exceptions, for example), and this is a question for future work.

8.2.1 *Role in Design of* DALI

The immediate technical benefit of this result is providing technical justification for using the reductions as semantics-preserving optimizations in an implementation. But there are other benefits that we are interested in from the point of view of language design:

1. It provides us with a basic understanding of the notion of observational equivalence. In particular, in the case of this language (as is in the case for many deterministic languages), one arrives at a simple equational theory simply be changing reduction arrows into "convertibility" equalities.

2. Taha's development[15; **?**] emphasizes *partitioning* expressions into values, workables, and stucks, and establishing "monotonicity properties" from which, for example, Wright and Felleissen's "Uniform Evaluation" [20] follows. Thus, not only do we provide the basis for posing the question of "what is a type system for datatypes with binder", we already provide some of the technical properties needed in establishing type safety for *any type system* that we may wish to investigate.

3. Attaining this result involves constructing a number of variations of the operational semantics, and relating them formally. This process provides a substantial amount of cross-checking between various definitions, and gives a very accurate operational understanding of the kind of invariants that a type system will be expected to guarantee.

9. RELATED WORK

DALI is a functional meta-programming language, and is related as such, to many other meta-systems.

Meta-systems built with a functional programming base include MetaML [**?**; 18], $\lambda^{\Box}$[5] and $\lambda^{\bigcirc}$[4]. These differ from DALI in that they are homogeneous systems, where the meta- and object-languages are the same. None of these systems provide mechanisms for analyzing the structure of object-programs.

Theorem prover based meta-systems have been constructed for several kinds of logics. Implementations of classical logics include the HOL [?] theorem prover, Isabelle [?], and the Prototype Verification System (PVS) [?]. Implementations of constructive (or intuitionistic) logics include Elf [?], Coq [?; ?], Nuprl [?] and Lego [?].

Finally, there are logic programming languages with meta-programming extensions, λ-Prolog [?; ?; ?], and L_λ [?]. These are prolog-like languages with extensions for representing and analyzing object-programs whose representations are based on the λ-calculus.

Of these systems, Isabelle, Elf, λ-Prolog, and L_λ use some sort of higher-order abstract syntax to represent object terms. Of these, all but L_λ, use higher order unification to implement intensional analysis of object terms. Higher order unification is in general undecidable, and does not guarantee a most general unifier.

L_λ implements a subset of lambda-Prolog, where intensional analysis is syntactically restricted to a form which is decidable using unification on higher-order patterns. It is this idea transferred to the functional programming world that is the basis for ML_λ and DALI.

The term *higher-order abstract syntax* was originated by Pfenning and Elliott [12]. This work provided a basis for automating reasoning in LF[6], and was used as the basis for the implementation of Pfenning's Elf[?] and its successor Twelf[?].

9.1 DALI vs. ML$_\lambda$

Dale Miller [?] describes ML$_\lambda$, a proposal to extend ML to handle bound variables in data-types. The idea of representing object-level bindings, in a functional language, using a binding construct different from the function abstraction of the meta-language derives from this paper. While our work takes Miller's proposed extensions as its basis, there are some differences:

- We distill the main ideas of Miller's ML$_\lambda$ into a basic calculus of core DALI. We concentrate on the the reduction semantics and equational theories of such a language. To the authors' knowledge, this work is the first instance of a sound reduction semantics for a functional language supporting binding constructs in data-types.
- We abandon the notion of *function extension* that allows extending the domain of arbitrary ML functions within the scope of an object-level variable. We find function extension needlessly difficult to model in a reduction system, and seek to introduce an alternative construct: patterns that match object-level variables. We conjecture that, together with equality over object-level variables, one can circumvent function extension without loss of expressiveness or good programming style.
- We abandon the notion of object-level application [?]. Rather, pattern matching on object-level bindings binds higher-order pattern variables to functions that perform appropriate substitutions directly, thus further simplifying formal development, and, in practical terms, internalizing object-level variable substitution, which in [?] must be defined separately for each data-type.

  However, internalization of such object-level substitution in presence of extensional function values is not without cost: we had to resort to a fine distinction between ground (or equality) values and the more standard notion of values in such calculi, and adjust evaluation and reduction to restrict the analysis of object-language terms to preserve soundness and confluence of the calculus.

DALI differs from most of the other work discussed above in following ways:

- It is functional and deterministic, and is presented as an extension of a standard CBV functional language. It provides support for higher-order syntax by providing a small number of new language constructs.
- The formal properties we have proven about the language suggest that the new features integrate well with the host functional language.
- The reduction semantics we provide gives rise to a simple equational theory that can be used to reason about program equivalence.

10. CONCLUSIONS AND FUTURE WORK

In this paper we have shown that a functional programming language with support for higher order abstract syntax through an additional object-level binding construct can be assigned a simple big-step semantics. We have defined a reduction semantics and presented important results of confluence and soundness w.r.t. evaluation of this reduction semantics for DALI. After this initial success much work remains to be done. In particular:

- Developing a basic type system for DALI. In addition to the traditional notions of safety there are some efficiency concerns that we expect that a type system could be used to alleviate. In particular, the discharge operation and the use of the ground-value restriction b in the semantics would incur significant run-time penalties in an implementation. We expect that an appropriate type system could help avoid these.
- Integrating with multi-stage programming. In particular, DALI meta-programming utility is orthogonal to that of multi-stage programming [17; 16; 16; 9; 2]: with DALI, the object language is allowed to vary, and intensional analysis is supported. Note, however, that DALI does support the hygienic synthesis of object code, although in a manner less concise than those of multi-stage programming languages. Finally, whereas it has been demonstrated that the former can guarantee that the synthesized code is type correct, the only guarantee that we have at the moment with DALI is that the synthesized code is syntactically correct.
- An implementation of a full programming language environment based on DALI. Although a full implementation of DALI is missing at the moment, the mechanisms of higher-order pattern matching and analysis

of object-level bindings has been implemented by Tim Sheard as an experimental feature of the MetaML interpreter [?].

From the point of view of semantic language design, in reproducing Taha's technical development of MetaML, we have found that all the proofs could be carried out in a systematic manner for the (considerably larger) language at hand, and that many of the proofs remain literally unchanged. This seems to be primarily due to the use of the notion of "workables" in parameterizing the various lemmata. In future work, we intend to investigate the extent to which this development can be generalized.

11. REFERENCES

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[2] Zine El-Abidine Benaissa, Eugenio Moggi, Walid Taha, and Tim Sheard. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)*, July 1999. To appear.

[3] R. M. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In *Proceedings of Advanced Course on Abstract Software Specifications*, pages 292–332. Springer-Verlag, Lecture Notes in Computer Science, 1980.

[4] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, July 1996. IEEE Computer Society Press.

[5] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 258–270, St. Petersburg Beach, January 1996.

[6] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings Symposium on Logic in Computer Science*, pages 194–204, Washington, June 1987. IEEE Computer Society Press. The conference was held at Cornell University, Ithaca, New York.

[7] Neil D. Jones, Carsten K Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993. Avaiable online from http://www.dina.dk/ sestoft.

[8] E. Moggi. Functor categories and two-level languages. In *FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[9] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[10] Eugenio Moggi. A categorical account of two-level languages. In *Mathematics Foundations of Program Semantics*. Elsevier Science, 1997.

[11] Emir Pašalić, Tim Sheard, and Walid Taha. DALI: An untyped, CBV functional language supporting first-order datatypes with binders (technical development). Technical Report CSE-00-007, OGI, March 2000. Available from [?].

[12] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, June 1988.

[13] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. Ergo Report 88-065, Carnegie Mellon University, Pittsburgh, July 1988.

[14] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing through staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, January 1998.

[15] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999. Revised October 99. Available from author (taha@cs.chalmers.se).

[16] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, July 1998.

[17] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997. An extended and revised version appears in [18].

[18] Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000. In Press. Revised version of [?].

[19] Masako Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, April 1995.

[20] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.