Extending Dylan's Type System for Better Type Inference and Error Detection

Hannes Mehnert *
IT University of Copenhagen, Denmark
hame@itu.dk

Abstract

Whereas dynamic typing enables rapid prototyping and easy experimentation, static typing provides early error detection and better compile time optimization. Gradual typing [26] provides the best of both worlds. This paper shows how to define and implement gradual typing in Dylan, traditionally a dynamically typed language. Dylan poses several special challenges for gradual typing, such as multiple return values, variable-arity methods and generic functions (multiple dispatch).

In this paper Dylan is extended with function types and parametric polymorphism. We implemented the type system and a unification-based type inference algorithm in the mainstream Dylan compiler. As case study we use the Dylan standard library (roughly 32000 lines of code), which witnesses that the implementation generates faster code with fewer errors. Some previously undiscovered errors in the Dylan library were revealed.

1. Introduction

Dylan is a dynamically typed object-centered programming language inspired by Common Lisp and ALGOL. Concepts inherited from Common Lisp are: methods are first class objects (generic functions), classes, multiple inheritance, multiple dispatch, keyword arguments, variable-arity methods, optional type annotations and macros (which are less powerful). Dylan's syntax is inherited from ALGOL, thus it does not have S-expressions, but an infix syntax and explicit `end` keywords. In contrast to Common Lisp, Dylan has a separation of compile time and runtime.

The abstraction λ `x.x` is written `method(x) x end` in Dylan. A literal list is written `#()` (the empty list): `#(1, 2, 3)` is the list with 3 elements, 1, 2 and 3. The motivating example to enhance Dylan's type inference is `map(method(x) x + 1 end, #(1, 2, 3))` which applies the anonymous method `x + 1` to every element of the list `#(1, 2, 3)`. Previously the compiler called the generic function `+`, since it could not infer precise enough types, using the type inference algorithm described in [2].

By introduction of parametric polymorphism (type variables) the types can be inferred more precisely. The former signature of our `map` is `<function>, <collection>` $\Rightarrow$ `<collection>`. A more specific signature using type variables would be $\texttt{<function>}_{\alpha\rightarrow\beta}$, $\texttt{<collection>}_{\alpha} \Rightarrow \texttt{<collection>}_{\beta}$, where the first parameter is a `<function>` which is restricted to $\alpha \rightarrow \beta$, the second parameter is a `<collection>` of α, and the return value is a `<collection>` of β. Using this signature, α will be bound to `<integer>`, and the optimizer can upgrade the call to `+` to a direct call to `+ (<integer>, <integer>)`, since the types of the arguments are `<integer>` and `singleton(1)`.

A problem that emerges is how to deal with a value whose type is not known until runtime? An initial solution would treat it as the top of the subtyping hierarchy, which would be `<object>` in Dylan. A problem is that then too few programs are accepted by the type checker, namely `method(x) successor(x) end` where `successor` takes an `<integer>`, would be rejected. A solution for this would be to allow implicit down-casts from `<object>` to anything. But then too many programs would be accepted, namely `(method(x) successor(x) end)(#t)`, since `#t` can be up-casted to `<object>` and be down-casted to `<integer>`, but would result in a runtime error.

Another solution for dynamic types is gradual typing [26], which is a formalized type system based on the typed λ-calculus with a specific type **dynamic**, which is unknown at compile time. Further extensions of gradual typing are for objects [27] and a unification-based type inference algorithm [28].

Dylan is well-suited to gradual typing, since its syntax already contains optional type annotations and Dylan's design contains devices to restriction of dynamism. But Dylan provides challenges for gradual typing, namely a nominal type system with subtyping, generic functions, multiple and optional parameters, multiple return values and variable-arity methods; solutions to these will be presented in this paper. A formalized type system is not accomplished, thus no safety proofs are given.

Typed Scheme [32] (now known as Typed Racket) already anticipated several achievements also presented here, namely union-based subtyping (occurence typing), integration of multiple and optional parameters, multiple return values and variable-arity polymorphism. In contrast to Typed Racket, which has Java-style first-class classes, Dylan has generic functions and multiple dispatch.

The main goal of gradual typing and Typed Scheme is to have interlanguage migration of a safe language whose typed portion is also type-safe, whereas gradual typing has a finer level of granularity in this respect.

Success typing [17] is a similar approach to the presented one, using type inference for error detection. Success typing was developed for Erlang and is done after the compiler inserted dynamic type checks where needed. The inference algorithm finds code locations where types are definitively disjoint and presents these to the user. The goal was to provide documentation and error messages at compile time of unmodified Erlang code.

* Work conducted as diploma thesis at TU Berlin, Germany

ILC'10 October, 19-21, Reno, Nevada, USA.

This paper is based on our diploma thesis [20] written at TU Berlin, Germany.

Contributions In this paper we show the practicability of gradual typing by extending it for a real-world programming language. Prior to this work Dylan supported selection of a method of the generic function by only first-order methods, this was extended to higher-order methods. Also we implemented type inference and describe the results compared to the previously implemented algorithm [2].

Structure The paper is structured as follows: The next section 2 gives an overview of Dylan and gradual typing. In section 3 we develop Dylan extensions for gradual typing. Afterwards, in section 4, we present the results of this approach, using the Dylan library as a case study. We relate the work in section 5 and finally conclude and show directions of further work in section 6.

2. Dylan and gradual typing

2.1 Overview of Dylan

Dylan was originally developed by Apple Cambridge, Carnegie Mellon University and Harlequin in the early 1990s. In 1996 the Dylan Reference Manual [25] was published, which is the language standard. Although the main influences of Dylan's design are Common Lisp, the Common Lisp Object System and Smalltalk, Dylan has an ALGOL like syntax.

There are two Dylan compilers available as open source, both implemented in Dylan: Gwydion Dylan, formerly developed at CMU, compiles Dylan to C. And Open Dylan[1], formerly developed at Harlequin and Functional Objects, available as open source since March 2004. Open Dylan compiles either to x86 assembler or to C. Open Dylan provides an IDE, including profiler, debugger and has thread support. This paper extends the Open Dylan compiler. The code can be found in the Open Dylan subversion repository as the `typed-opendylan` branch[2].

In Dylan methods are first class objects and multiple methods with the same name can be specified, which need to vary in at least one argument type. The selection of the most-specific method, which is actually called, is based on all required arguments. This concept is named *multiple dispatch*. According to [22] multiple dispatch is profitable, since in languages which only provide single dispatch the double dispatch design pattern [9] is used. The latter is based on runtime type introspection (`instanceof`) and leads to runtime type checks which decreases runtime speed and gives fewer opportunities for static analysis. Multiple dispatch also solves the expression problem [33] in an elegant way.

Because the most-specific method has to be selected at every call-site at runtime, generic function dispatches pose a major performance drawback. To accomplish finding the most-specific method, the distance between the formal and actual type of every argument is computed, for every method of the generic function. This is a time consuming computation, increasing with the amount of arguments and defined methods.

In order to allow some optimizations, a concept named *sealing* [25, Ch 9, Sealing] was introduced in Dylan. This allows to restrict extensions of generic functions in subsequent libraries or at runtime, as well as to protect parts of the class hierarchy from being further extended. For example the generic function `+` can be extended by normal developers, but it is sealed on `<integer>, <integer>` and subclasses thereof to enable compilers to inline calls to `+` with two `<integer>` arguments, resulting in a direct call of the method instead of a generic function dispatch.

[1] http://www.opendylan.org

[2] svn://anonsvn.opendylan.org/scm/svn/dylan/branches/typed-opendylan

Other features present in Dylan are exceptions, as found in Common Lisp. Methods with optional keyword arguments can be specified as well as variable-arity ones. Every method is compiled into binary code which contains then multiple entry points [19], one which reorders the keyword arguments and removes the keywords, another which expects the arguments to be in the correct order on the stack.

Dylan also supports multiple inheritance, with a superclass linearization algorithm specified in [25, Ch 5, Types and Classes]. This does not provide method monotonicity, but an algorithm with monotonicity is described in [4], which would allow compression of generic function dispatch tables [14]. With a linearization algorithm a class precedence list is built, preventing problems that a member is inherited multiple times, which might be the case in other programming languages like C++.

The object initialization is similar to the one in Common Lisp, a method `make` is called by the user (`new` in other languages), which allocates space for the object, and initializes the slots (members, fields). Once the space is allocated, the method `initialize` is called, which can be customized for any class to do something special like registration in a global table.

The type inference algorithm [2] is based on call caches, whenever a generic function is called with a distinct set of argument types, the return type of this generic function is cached at compile time, in order to allow more precise type inference than just using the most generic return value type.

Type Safety In statically typed languages type safety is defined such that well-typed programs can't go wrong. A sound type system is not lenient, in that it rejects all invalid programs and some valid programs. A complete type system accepts all valid programs and some invalid programs. In a dynamically typed language like Dylan, the demand to the type system is completeness; dynamic type checks are emitted where needed. If a program does not contain any dynamic type checks or runtime generic function dispatches, it can't go wrong and is thus sound and complete.

Differences between Dylan and Common Lisp Initially a prefix syntax (using S-Expressions) was specified for Dylan, but that syntax was dropped in [25] to attract developers from the C++ community. Because the syntax contains more structure, the macro system is different than the one of Common Lisp. The macro system is based on pattern-matching and is hygienic (but allows explicit unhygienic parts), and is less powerful than the Common Lisp macro system. An extension [3] has been proposed which advances the macro system to be as powerful as Common Lisp's. This extension is implemented in and used by the Open Dylan compiler.

Other differences between Common Lisp and Dylan are that Dylan has a clear separation of compile time and runtime. This implies that there is no need to have a compiler available at runtime and there is no `eval` function. Also, Dylan emphasizes safety by providing no possibility to access private members. And there is also no way to force the compiler to remove bounds checks in potentially unsafe places (opposed to Common Lisps `safety 0`). Dylan also specifies sealing, described above in this section, which restricts dynamism and allows static analysis.

2.2 Gradual Typing

In this subsection we give an overview of Siek's and Taha's work with some examples of gradual typing [26].

Gradual typing is a concept developed by Siek and Taha [26] to support a smooth migration from dynamically to statically typed code. It extends the simply-typed λ calculus ($\lambda_{\rightarrow}$) with a type representing the dynamic type, unknown at compile time, denoted by $?$. The gradually-typed λ calculus is written $\lambda^{?}_{\rightarrow}$.

$$\gamma \sim \gamma$$

$$\tau \sim ?$$

$$? \sim \tau$$

$$\frac{\tau_3 \sim \tau_1 \qquad \tau_2 \sim \tau_4}{\tau_1 \to \tau_2 \sim \tau_3 \to \tau_4}$$

Figure 1. Type consistency relation $\sim$

The main idea of gradual typing is the notion of a type whose structure may only be partially known. The unknown portions are indicated by ?. An example is the type ($number \times ?$), which represents a tuple type whose first element is a $number$, and whose second element has an unknown type. Programming in a dynamically typed style can be done by omission of type annotations, which then get assigned the type ?. Type annotations can be added to facilitate more compile time type correctness by the type checker, possibly with ? inside the types to retain some flexibility.

A static type system should reject programs that have inconsistencies in the known parts of types. For example, the program ((λ (`x` : $number$) (succ x)) `#t`) should be rejected because the type of `#t` is not consistent with the declared type of argument `x`. That is, $boolean$ is not consistent with $number$. On the other hand, the program ((λ (`x`) (succ x)) `#t`) should be accepted by the type system, because the type of `x` is considered unknown. A type error will be raised at runtime by the application (succ `#t`).

Since type equality with ? is not sufficient, the type consistency relation ($\sim$) is introduced and axiomatized with the definition in figure 1. It is reflexive and symmetric, but not transitive. In the following, τ will be used as a variable for any type, while γ will be used for ground types, like $number$ or $boolean$. Some examples where the type consistency relation holds are: $number \sim number$, $number \sim ?$, $number \to boolean \sim ?$, $number \to ? \sim ? \to number$. But the following examples are not type consistent: $number \to number \not\sim number \to boolean$, $number \to ? \not\sim boolean \to ?$.

In [26] the relation of $\lambda^{?}_{\to}$ to the untyped λ calculus is presented, together with a translation which converts any λ term into an equivalent term of $\lambda^{?}_{\to}$. It is not possible to accept all terms of the untyped λ calculus and provide type safety for full-annotated terms, because the ill-typed terms are not accepted. An example is (`succ #t`), which is a valid term in the untyped λ calculus, but since `succ` is of type $number \to number$, and `#t` is a $boolean$, it is not accepted.

The relation of $\lambda^{?}_{\to}$ to the simply typed λ calculus ($\lambda_{\to}$) is that both calculi are equivalent for terms of the $\lambda_{\to}$, proven in [26, Theorem 1]. A direct consequence of the equivalence is that the gradual type system catches the same static errors as $\lambda_{\to}$.

Subtyping In [27] subyping is integrated into gradual typing. That paper extends the object calculus ($Ob_{<:}$) of Abadi and Cardelli [1] with the dynamic type. There are two kinds of subtyping, nominal and structural [23, Chapter 19]. While in a nominal system a developer explicitly writes down the superclasses of a class (as done in mainstream object-oriented languages like Java, Common Lisp, Dylan), in a structural system the subtyping relation is defined on the structure of types (as done in Ocaml, JavaScript, F#) and if the members are equivalent, the types are equivalent. Gradual typing was applied to a structural type system. Aldrich [18] worked on integration of structural and nominal subtyping.

The subtype relation is denoted by $<:$. We already motivated why the dynamic type allows implicit down-casts in the introduction. If ? is treated as the top of the subtyping hierarchy, the hierarchy would end up in a single type. Since the down-cast $? <: S$ and the standard up-cast $R <:?$ holds, applying the regular transitivity rule $R <: S$ would hold for all types R, S and the type checker would not reject any program.

For that reason the dynamic type ? is treated neutral to subtyping, the single subtyping rule where ? applies is reflexivity, $? <: ?$. A type system for $Ob^{?}_{<:}$ (the gradual object calculus) is defined in [27], as well as a type checking algorithm, operational semantics and safety proofs.

$$? \sqsubseteq \tau$$

$$\gamma \sqsubseteq \gamma$$

$$\frac{\tau_1 \sqsubseteq \tau_3 \qquad \tau_2 \sqsubseteq \tau_4}{\tau_1 \to \tau_2 \sqsubseteq \tau_3 \to \tau_4}$$

Figure 2. $\sqsubseteq$ relation

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_g x : \tau | \{\}} \text{ CVAR}$$

$$\Gamma \vdash_g c : typeof(c) | \{\} \text{ CCNST}$$

$$\frac{\Gamma \vdash_g e_1 : \tau_1 | C_1 \qquad \Gamma \vdash_g e_2 : \tau_2 | C_2 \quad (\beta\ fresh) \quad C_3 = \{\tau_1 \simeq \tau_2 \to \beta\} \cup C_1 \cup C_2}{\Gamma \vdash_g e_1 e_2 : \beta | C_3} \text{ CAPP}$$

$$\frac{\Gamma(x \mapsto \tau) \vdash_g e : \rho | C}{\Gamma \vdash_g \lambda x : \tau . e : \tau \to \rho | C} \text{ CABS}$$

Figure 3. Constraint generation for $\lambda^{?\alpha}_{\to}$

2.3 Unification-based inference

In [28] a type inference algorithm for $\lambda^{?\alpha}_{\to}$ (the polymorphic gradually typed lambda calculus) is presented, which we will summarize here. First, the relation *less or equally informative*, written $\sqsubseteq$, is presented in Figure 2. This relation is the partial order underlying the $\sim$ relation presented in the previous section.

The main idea of the presented algorithm is that for each type variable α a type τ is maintained, which is the lower bound on the solution of α. When another constraint $\alpha \simeq \tau^*$ is encountered, the lower bound is moved up to the least upper bound of τ and τ^*.

Applying this to the previous example, $\alpha \to \alpha \simeq ? \to number$, α gets first the type ? assigned. Next the least upper bound of ? and $number$ is computed, which is $number$. Thus, α gets the type $number$ assigned.

This idea is integrated into Huet's almost linear algorithm [13]. This works in two separate phases, the first generates constraints, the second solves the constraint set. The algorithm uses a graph representation, where every type variable as well as each type is a node. There are two kinds of edges, constraint edges representing constraints, and graph edges from composite nodes (like $\to$) to their children.

Constraint Generation The constraint generation judgement has the form $\Gamma \vdash_g e : \tau | C$, where C is the set of constraints. In figure 3 the constraint generation rules are given. The rules are tightly

connected to the type system, equivalence is shown in [28, Lemma 3]. The only rule which actually generates a constraint is CAPP between the type of the function expression and the type of the argument $\rightarrow$ a fresh type variable.

Constraint Solver The definition of the algorithm `solve` is shown in listing 1. Its input is a set of constraints (line 1). Description of `copy_dyn` and `contains_vars` is deferred to later. In each iteration (lines 6-19) the algorithm takes a constraint from the set (line 6), looks up the representatives of both sides of the constraint (line 7) and then orders (line 9) the nodes and merges the equivalence classes of the nodes (line 10). Afterwards a case analysis is done on the real types (`type`) of the nodes (lines 11-20), where possibly more constraints are added. If the constraint set is empty, the quotient graph by equivalence class is constructed (line 21). An equivalence class contains all nodes which share a common representative. If the quotient graph is acyclic, a mapping from each node u in the original graph to the type of its representant is returned (line 23). Otherwise an error is returned (line 24).

Code listing 1 Huet `solve` algorithm

```
    Input: Constraints C
    C := copy_dyn(C)
    for each node u do
        u.contains_vars := true
 5: while not C.empty
        (x ≃ y) := C.get
        u := find(x); v := find(y)
        if u ≠ v then
            (u, v, f) := order(u, v)
10:         union(u, v, f)
            case type(u) ≃ type(v) of
                u1 → u2 ≃ v1 → v2 ⇒
                        C.add(u1 ≃ v1); C.add(u2 ≃ v2)
                u1 → u2 ≃ ? ⇒
15:             if u.contains_vars then
                    u.contains_vars := false
                    C.add(vertex(type=?, contains_vars=false) ≃ u1)
                    C.add(vertex(type=?, contains_vars=false) ≃ u2)
                τ ≃ var | τ ≃ ? | γ ≃ γ ⇒ pass /* do nothing */
20:             _ ⇒ error: inconsistent types
    G = quotient graph by equivalence class
    if G acyclic then
        return { u ↦ type(find(u)) | u a node in original graph }
    else error
```

When two nodes are merged, a decision has to be made which node overrides which other node. This is based on the less or equally informative relation shown in figure 2. The decision is done in the relation `order`, shown in listing 2. A type variable α is overridden by any other type (lines 2-4). The dynamic type ? overrides type variables (line 2), but is overridden by any other type (line 3). The flag returned indicates whether the order of the nodes is relevant or might be changed by `union`.

Code listing 2 Relation `order`

```
order(u, v) = case type(u) ≃ type(v) of
    ? ≃ α ⇒ (u, v, true)
    ? ≃ τ | α ≃ τ ⇒ (v, u, true)
    τ ≃ α ⇒ (u, v, true)
    _ ⇒ (u, v, false)
```

The actual merging is done in `union` (listing 3), which takes two nodes and a flag as input (line 1). If the flag is true, the first node is more informative and is chosen as representative (lines 2-5). The rank is then only increased if the rank of both nodes is equal (lines 3-4). If the passed flag is false, the node with higher rank is chosen as representative (lines 7, 9), as in a regular union-find algorithm. Again, the rank is only increased if the rank of both nodes is equal (lines 10, 11).

Code listing 3 Huet `union`

```
    union(u, v, order_matters?) =
 2:     if order_matters? then
            if u.rank = v.rank then
 4:             u.rank := u.rank + 1
            v.representative := u
 6:     elseif u.rank ¿ v.rank then
            v.representative := u
 8:     else
            u.representative := v
10:         if u.rank = v.rank then
                v.rank := v.rank + 1
```

The case analysis of `solve` (lines 11-20) consists of several cases:

- The first case (line 12-13) matches when two function types are constrained. Arguments and values of both function types are constrained separately and added to the constraint set.
- The second case (line 14-18) matches if a function type is constrained to the dynamic type. Two constraints are added, both arguments and values of the function type are constrained to a newly allocated dynamic type each (lines 17 and 18).
- The third case (line 19) catches all other valid constraints, namely any type constrained to a type variable or the dynamic type, or a constraint consisting of two equivalent ground types. Nothing is done for these constraints.
- If none of those cases matches, an error is reported (line 20).

Consider the program (λ `f` : ?. λ `x` : α . f x), where the constraint $? \simeq \alpha \rightarrow \beta$ is generated, β is the result of the application. The type of `f` is transformed to $? \rightarrow ?$, thus we have the constraint $? \rightarrow ? \simeq \alpha \rightarrow \beta$, which is then solved to the constraints $? \simeq \alpha$ and $? \simeq \beta$. If the type of α is $\alpha \rightarrow \alpha$, an infinite loop would occur in the solver (by replacing $? \simeq \alpha$ with $? \rightarrow ? \simeq \alpha \rightarrow \alpha$, which is solved to $? \rightarrow \alpha$ twice, etc.). To prevent this infinite loop, each node has a flag `contains_vars`, which is initialized to true at the beginning (line 3-4) and set to false when the second case is matched.

The `copy_dyn` function replaces each ? with a new copy of ?, removing any sharing between ? nodes. Consider the program from listing 4, which generates (by applying rule CAPP twice) the constraint set $\{ number \rightarrow ?_0 \simeq ?_0 \rightarrow \beta_0, boolean \rightarrow ?_0 \simeq ?_0 \rightarrow \beta_1 \}$ in the last line, where $?_0$ is a single node. It is used as return value of `f` and `g` and as type of `a`. The type variables β_0 and β_1 are freshly allocated by the CAPP rule, and represent the return value of `f` and `g`. Solving these constraints results in the contradicting constraints $number \simeq ?_0$ and $boolean \simeq ?_0$. Thus, an error is reported. To prevent this error, the function `copy_dyn` replaces all $?_0$ with separate nodes, which can then be unified with different types.

Code listing 4 Conditional, `a` has distinct types in branches

```
let z = ...
let f (x : number) = ...
let g (y : boolean) = ...
let h (a : ?) = if z then f a else g a
```

The quotient graph is constructed by adding each node of the type graph which is its own representative (all nodes are initialized

to be their own representative). Then each edge of the type graph which originates in a representative node is put into the quotient graph. If the quotient graph contains a cycle, an error is reported (line 24). Otherwise the mapping of type variable to type is returned (line 23). An example where the quotient graph contains cycles is λ x . x x. The type of x is constrained with a function type (due to the application) whose argument is the type of x.

3. Dylan extensions of gradual typing

The type system needs to support the wide variety of type constructors defined by the Dylan language. The type constructors will be discussed in this section, as well as the extensions to Dylan and the extensions for the type inference algorithm.

3.1 Type constructors

As already mentioned, Dylan is a class-based programming language with a nominal type system. Thus each definition of a class leads to a type where the supertypes are explicitly written.

But as opposed to some mainstream object-oriented programming languages (like Java), not every type is a class, but there are other constructors.

Method definitions define the number of required arguments and their types, as well as optional keyword arguments and types, and return types. The types of required arguments and return values correspond to a tuple type in type theory, while the optional arguments form a record type. There are variable-arity arguments, which need special treatment, a *tuple-with-rest* type is used.

Singleton types contain a single runtime value, like `singleton(#f)` is the type of `#f`. Singleton types are for example useful in the instantiation process of Dylan, namely for the method `make`, where no instance is yet available, to extend this method with specific behaviour for custom classes.

Union types, constructed by `type-union`, a function that takes any number of types and returns a type which consists of the union of all given types. This is useful in several places, for instane in a method which might not succeed, as `resolve-host-name`, which translates a host name into an internet address. The return value of this method is `type-union(<internet-address>, singleton(#f))`, thus `#f` is returned if the host name cannot be resolved. In other programming languages (like C, Java) special values (`null`, -1) have to be reserved and the result value has to be tested for those special values, while in Dylan the type checker can use the information that a specific value might be returned apart from the normal type. Functional languages have the polymorphic `Option[`α`]` or `Maybe[`α`]` type, which encapsulates either a value or `None` or `Some[`α`]`. This bears the inconvenience that to get to the value, it first needs to be unpacked.

Limited types [25, Ch 5, Limited Types], constructed by the function `limited`. Limited types in Dylan constrain a base type. There are only three different base types which might be constrained, `<collection>`, `<integer>` and `subclass`. The element type of a collection and its length can be constrained. Integer ranges can be defined, for example the type of all numbers between 0 and 16, written `limited(<integer>, min: 0, max: 16)`. Also, the type of a class and all subclasses can be specified by `subclass(<number>)`, which contains all subclasses of `<number>`. This is useful for constructors of a class and its subclasses.

3.2 Extensions to Dylan

We implemented two extensions for type constructors in Dylan: polymorphic type variables and function types.

Previously the only expressible type of a function was `<function>`, which does not specify the number and types of arguments and return values. In order that a developer is able to have more fine-grained control, we introduced new syntax for function types. For example the type expression `<string> => <integer>` specifies a function which has a single argument of type `<string>` and returns a single `<integer>` value.

The other extension are type variables. An example is the type of the identity function as `(forall: A) (x :: A) => (y :: A)` (was `(x :: <object>) => (y :: <object>)` previously). Type variables are written in parens just before the argument list, which is backwards-compatible to existing Dylan code. This enables a developer to write more expressive types, and allows the type inference to infer more concise types. Looking at the motivating example of this paper, the type of `map` can not be specified such that the type of the given function depends on the type of the given collection, resulting in more optimization opportunities.

3.3 Extended type inference

The type inference is extended with some language constructs like loop, conditional and multiple value handling, as well as some more interesting types, like tuple types, record types and singletons. The former extend the constraint generation, while the latter extend the constraint solution phase.

Assignment A variable which is mutated is converted to a reference cell. The constrained type for a cell is the union of all assigned values.

Loop Type inference of a loop is done initially with the type of all loop variables from the outer scope. If after inference the resulting type of the loop variable is equal to the type of the loop variable in the outer scope, this type is used. Otherwise the top type is used, since otherwise safety might be violated.

Conditionals In order to support conditionals, different type environments are setup for each branch of the conditional. In each branch then the type of a binding can be made more specific.

Multiple values There are two intermediate language constructs, one for constructing a multiple return value, one for extracting a single element out of a multiple value. The former constructs a tuple type and constrains this with all values which are put into the multiple value. The latter constrains the indexed value from the tuple type with the resulting single value.

Singleton At various points during type inference, for example when a literal list is encountered, a singleton type is not used, but rather the supertype of the singletons. The *most-specific* type of the literal list `#(1, 2, 3)` would be `singleton(#(1, 2, 3))`, but the more general `limited(<list>, of: <integer>)` is inferred instead. Otherwise the compiler has to do a lot of useless work in subtype-tests. But the question of how specific types should be inferred is still open. Success typing [17] uses a threshold value of four unionees to convert it to a more general type.

Solve algorithm The solve algorithm is extended with additional composite types and subtyping. The extended `union` is shown in listing 7, `solve` in listing 8.

The solution strategy for composite types is propagation to their children. Both types must be of same structure, thus a tuple type ($\times$) cannot be unified with a function type ($\rightarrow$). A special case is variable-arity tuple type, which, if unified with a tuple generates new children nodes on demand. A variable-arity tuple is always less specific than a tuple. This affects both `union` and `solve`.

Instead of using type consistency in `solve`, the subtype test is used. Thus, if two types, `<number>` and `<integer>` are unified, `<integer>` is used as representative, because it is more specific. This narrows down the types of data flow nodes. This might lead to type errors, for example in listing 5, where in line 9 `y` will be constrained to type `<number>`; in line 10 it will be constrained

to an `<integer>`, solving these constraints result in `<integer>`. Accordingly, another traversal of the flow graph is needed, which emits a type check after line 9 that `a` is actually of type `<integer>`.

Code listing 5 Type error during type narrowing

```
   define method a (x) => (y :: <number>)
2:    ...
   end;
4:
   define method b (x :: <integer>)
6:    ...
   end;
8:
   let y = a(42);
10: b(y)
```

The extended relation `order` is changed as shown in listing 6. The added lines are 5 and 6, which take care that a variable-arity type (denoted `rest`) is not used as representative of any other type, since it is less informative. But a variable-arity type is more informative than a type variable or the dynamic type (lines 2 - 4).

Code listing 6 Extended relation `order`

```
   order(u, v) = case type(u) ≃ type(v) of
2:     ? ≃ α ⇒ (u, v, true)
       ? ≃ τ | α ≃ τ ⇒ (v, u, true)
4:     τ ≃ α ⇒ (u, v, true)
       τ ≃ rest ⇒ (u, v, true)
6:     rest ≃ τ ⇒ (v, u, true)
       _ ⇒ (u, v, false)
```

The extended `union` function is shown in listing 7. The first case (lines 2-5), if `order` set the flag that order of arguments is relevant, is unchanged.

If either `subtype?(u.type, v.type)` or `subtype?(v.type, u.type)`, the more specific type is used as representative (lines 7-12). This alone is unsafe, since it does not result in the least upper bound, rather the more specific value is used as representative. This is safe because it is used together with the mentioned emitted runtime type checks when the result of an operation returns a broader type then the inferred for a data flow node.

If those three special tests did not succeed, the common case from the original algorithm is used, the node with higher rank is used as representative (lines 13-18).

The case statement of the extended solve algorithm shown in listing 8. The solve algorithm supports tuple types (lines 8, 9) and limited collections (line 15), both push constraints to their children, the same strategy used by → constraints.

4. Evaluation

The explained algorithm was implemented into the mainstream Dylan compiler. The standard Dylan library, including startup of the Dylan runtime environment, collections, etc. was used as a case study. The code size is approximately 32000 lines of code. In this library already several programming errors were discovered, which had not been discovered by the former type inference algorithm [2].

Parametric polymorphism was used in `make` and `as`, instead of having the type signature `(<type>, #rest arguments) => <object>` it now has the signature `(forall: A)(A == <type>, #rest arguments) => A`. The latter type signature is actually the documented behavior in [25, Ch 12, Constructing and Initializing Instances]: “Note that the `<class>` method on make returns a newly allocated direct instance of its first argument.”. The coercion method `as` is similar, it receives a type `t` and an object `x`, and should return an object of type `t`, which is `x` coerced to type `t` (according to the documentation).

Code listing 7 Extended Huet `union`

```
   union (u, v, order_matters?) =
2:    if order_matters? then
         if u.node − rank = v.node − rank then
4:          u.rank := u.rank + 1
         v.representative := u
6:    else
         if subtype?(u.type, v.type) then
8:          u.rank := max(u.rank, v.rank) + 1
            v.representative := u
10:      elseif subtype?(v.type, u.type) then
            v.rank := max(u.rank, v.rank) + 1
12:         u.representative := v
         elseif u.rank ¿ v.rank then
14:         v.representative := u
         else
16:         if u.rank = v.rank then
               v.rank := v.rank + 1
18:         u.representative := v
```

Code listing 8 Extended Huet `solve`

```
   case type(u) ≃ type(v) of
2:    u1 → u2 ≃ v1 → v2 ⇒ C.add(u1 ≃ v1); C.add(u2 ≃ v2)
      u1 → u2 ≃ ? ⇒
4:          if u.contains_vars then
               u.contains_vars := false
6:             C.add(vertex(type=?, contains_vars=false) ≃ u1)
               C.add(vertex(type=?, contains_vars=false) ≃ u2)
8:    u1 × ... × un ≃ v1 × ... × vn ⇒ C.add(u1 ≃ v1); ... ;
C.add(un ≃ vn)
      u1 ... un ≃? ⇒
10:         if u.contains_vars then
               u.contains_vars := false
12:            C.add(vertex(type=?, contains_vars=false) ≃ u1)
               C.add(vertex(type=?, contains_vars=false) ≃ un)
14:   lcoll_Cu(τU) ≃ lcoll_CV(τV) ⇒ C.add(CU ≃ CV);
C.add(τU ≃ τV)
      τ ≃ var | τ ≃ ? | γ ≃ γ ⇒ pass
16:   _ ⇒ error: inconsistent types
```

4.1 Interaction of parametric polymorphism and generic functions

Parametric polymorphism also affects generic function dispatch. The definition of method specificity and congruency has to be extended with parametric polymorphism in mind.

Intuitively, if a generic function is defined, only congruent polymorphic methods may be added. A polymorphic method is congruent if its signature with the broadest type of all type variables is congruent.

Method specificity of polymorphic methods depends on the actual arguments. During method dispatch, the type variables are bound to the types of the actual values. This idea has been proposed in [15].

A result of this definition is that if a method contains a polymorphic type variable at an argument position, this is the most specific at that position (within the upper bound). Methods which are specified on a subtype of the upper bound at the given argument position will never be more specific. In order to avoid this ambigu-

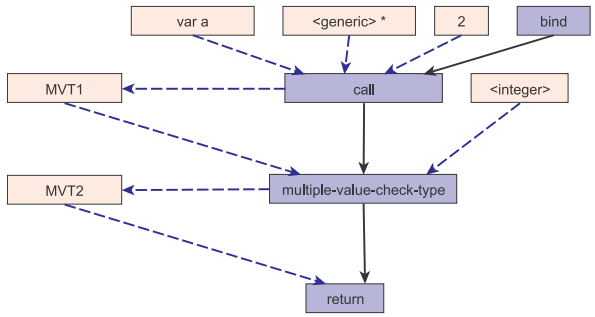

Figure 4. Flow graph of method `double`

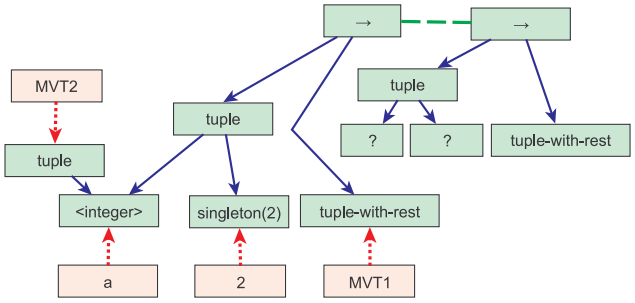

Figure 5. Type graph of method `double`

ity a method definition which has a subtype of a polymorphic type variable at any argument position is rejected.

Another matter are polymorphic generic functions, which are useful to specify a given protocol more precisely. Types of polymorphic generic functions can be inferred even if the actual method which is called is unknown at compile time. This is used for example for the methods `make` and `as`. Methods implementing polymorphic generic functions have to fulfill the instantiated polymorphism.

4.2 Example: Double

A simple example is the method `double`, whose code is shown in listing 9, and its flow graph is in figure 4. The bright nodes are data flow nodes, while the others are control flow nodes. The dashed edges are data flow edges, filled are control flow edges. `MVT` is a multiple value temporary, thus a data flow element with multiple values. The initial run of the type inference algorithm builds the type graph shown in figure 5.

Code listing 9 `method double`

```
define method double (a :: <integer>)
   => (result :: <integer>)
     2 * a
end
```

As already described, method upgrading of the generic function call of `*` succeeds, because the types of the arguments are subtypes of `<integer>` (namely `singleton(2)` and integer), and the method `*` is sealed in the domain `(<integer>, <integer>)`. The resulting flow graph is shown in figure 6. The inferred type graph is shown in figure 7. When this type graph is solved (as shown in figure 8), the optimizer can fold the type check, since the multi-value temporary `MVT1` is guaranteed to contain a single value of

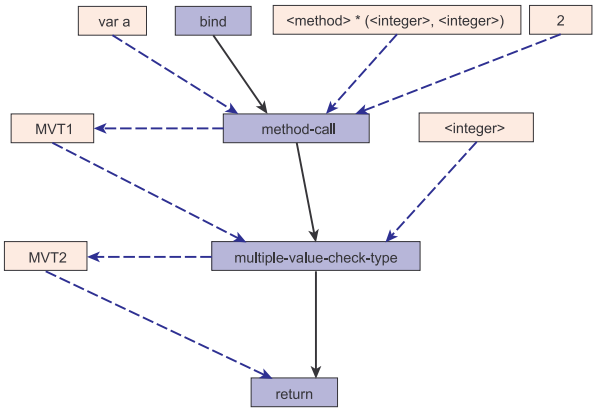

Figure 6. Flow graph of upgraded method `double`

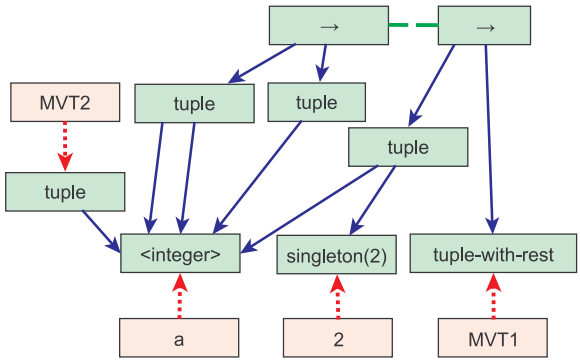

Figure 7. Type graph of upgraded method `double`

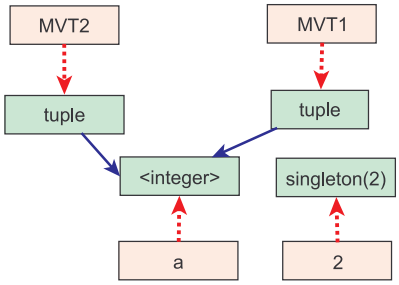

Figure 8. Final type graph of method `double`

type `<integer>`. The call to `*` is inlined and primitives are emitted.

4.3 Example: Map

Another example is the code snippet `map(method(x) x + 1 end, #(1, 2, 3))`. The method `map` has the signature `(forall: A B)(f :: A => B, l :: limited(<list>, of: A)) => (res :: limited(<list>, of: B))`.

The inference algorithm works as following: The result type, which is searched for, is named β. Due to the application, the constraint $\alpha \rightarrow \beta \simeq$ `(A → B × limited(<list>, of: A)) → limited(<list>, of: B)` is generated. All occurrences of `A` and

B point to the same node. The right-hand side is the type of map. The solver decomposes this constraint into two constraints, where alpha gets decomposed into two type variables ($\alpha_0 \times \alpha_1$), one for each part of the tuple type. This gets decomposed into the constraints $\alpha_0 \simeq$ A $\rightarrow$ B and $\alpha_1 \simeq$ limited(<list>, of: A). The type of the literal list #(1, 2, 3) is used to create the constraint $\alpha_1 \simeq$ limited(<list>, of: <integer>), thus limited(<list>, of: A) $\simeq$ limited(<list>, of: <integer>). This is solved by propagation to the element type, A $\simeq$ <integer>, which binds A to <integer>.

Now the constraint $\alpha_0 \simeq$ <integer> $\rightarrow$ B is solved. The type of the anonymous method x + 1 is inferred, where x is of type <integer>. Thus the call to + is upgraded, since + is sealed on (<integer>, <integer>), and thus cannot be extended in any further library. The result type of the specific + method is a single <integer>, which then binds B to an <integer>.

The last constraint is $\beta \simeq$ limited(<list>, of: B), where B is bound to <integer>, thus β is set to limited(<list>, of: <integer>).

4.4 Safety results

Some code in the Dylan library was *ill-typed*, where methods were called with arguments where they were not defined. An example is function-next?[3], which takes a <method> argument, but was called with a <lambda>, which is a supertype of <method>. This was not discovered by the former type inference algorithm, but this code was never called with a concrete <lambda> or a subclass of <lambda> disjoint from <method>.

Several monomorphic make and as method implementations of the *polymorphic generic functions* specified to return any object, not an object of the given type. This has been fixed in the Dylan library, since the definition of the generic function now enforces the return type to be the requested type.

A subtle problem was in the *optimizer of the compiler*. It did unsafe optimizations but upgrading boxed values to unboxed values, but retaining the boxed type information, particularly for loop variables. When any such loop was inlined in another method, and type inferred once more, the declared and expected type did not match.

The *unsafe control flow* node <guarantee-type> was eliminated, which allowed the user to inject a type for any object in Dylan code. This is a major safety issue, since the compiler trusts the user to provide only correct type information. It was in the compiler to obtain performance in places where the old inference algorithm did not return specific enough (to optimize generic function dispatches) types.

There were also several *ill-typed primitives* in the compiler, for example the primitive apply had declared to take an <integer> containing the number of arguments.

4.5 Performance results

More specific types are inferred for *loop variables*. The loop variable i in Listing 10 was previously inferred to be of type <object>, while it is now of type <integer>.

Code listing 10 Simple loop example

```
let i = 0;
while (i ¡ 42)
    i := i + 1
end
```

In the compiler a *copy-down* method was implemented, where define copy-down-method takes a method name and a list of arguments and copies the body of the most specific method of the generic function and specializes it on the given argument types. The reason for *copy-down* methods were performance by reducing the amount of generic function dispatches. By the implementation of the more general concept, parametric polymorphism, *copy-down* methods are superfluous and could be eliminated.

Since the performance of the implemented algorithm is currently too low, due to the lack of incremental type inference and interaction of type inference and optimization, a larger case study was not accomplished. Once this issue of incremental type inference has been fixed, a larger case study will be done, an appealing case study will be the compiler itself.

The presented implementation does not yet handle variable-arity polymorphism [30] and Dylan's list comprehension are defined with variable-arity. Thus there are no statistics yet, but our expectation is that this will decrease the number of generic function dispatches by an order of magnitude.

Using the current implementation and the mostly unmodified Dylan library (only introducing a polymorphic as and make) the improvement is roughly 1%, in both generic function dispatches and dynamic type checks. The former compiler emitted 2107 generic function dispatches and 1897 dynamic type checks, while the new type inference implementation emits 2093 generic function dispatches and 1882 dynamic type checks. This is already a good result taking into account that the old type inference implementation had several special cases for commonly used methods where the inference was not powerful enough to infer specific types. Additionally it allowed users to inject types into the inference. Our implementation does not allow type injection, and has no special cases for specific method names.

5. Related Work

Soft typing There are several approaches to add or use type information in untyped languages. Some assist programmers by reporting type errors at compile time, while others help compilers to optimize written code. Either developers have to add type annotations to code or types are inferred.

An early approach (1991) was soft typing, developed by Cartwright and Fagan [6]. The goal of the research agenda was, that a developer shouldn't have to write down any type annotations. The early soft typing systems inferred complex type expressions even for simple expressions, thus error messages were not easy to decipher.

Later soft typing systems improved error messages by using different inference algorithms, but the error discovery was still conservative. By providing explicit type annotations, those can be used to locate errors and provide good error messages.

Strongtalk [5] (1993) is a Smalltalk with a static type system. In contrast to the presented approach, Strongtalk is based on a structural type system and does not integrate type inference. Also, Smalltalk does not have a class-based object system.

Scheme There are several approaches in the Scheme community on extending Scheme with static types. Scheme is different from Dylan in several aspects, one is that no class system is specified.

Henglein developed already in 1991 in [11] a $\lambda_{\rightarrow}$ calculus with a **Dynamic** type and explicit coercions. Using this calculus he presented 1995 in joint work with Rehof [12] a Scheme to ML compiler, providing dynamic polymorphic type inference for Scheme.

A more recent approach is Typed Scheme [32] (now known as Typed Racket), which presents a type system for Racket. It integrates type annotations into Racket and polymorphic type variables. The unique feature of their type system is occurrence typing, which uses the type information of the test in a conditional in both branches. Racket code can quite easily be ported to Typed Racket

[3] in discrimination.dylan

code and it is also possible to use typed and untyped modules side by side, while the typed ones cannot be blamed [34]. The granularity of Typed Racket is on a module basis, while in gradual typing it is a finer level of granularity. Runtime type errors cannot originate from statically typed code regions. Racket is different from Dylan, it does not contain multiple dispatch but Java-style first-class classes.

Blame calculus Contemporaneously to Siek and Taha [29], Tobin-Hochstadt and Felleisen [31] introduced the idea of migrating untyped modules to typed modules, leaving the system in a mixed but sound state. Their proof mechanism exploits Findler and Felleisen's blame system; to be precise, they showed that if an error occurs (which also happens in soundly typed languages), then the blame falls on an untyped module. Wadler and Findler merely formulated a more elegant framework for this proof and gave it a name (blame calculus) [34].

Ruby An approach which integrates static types into Ruby is presented in [8]. A type system, type annotation syntax and a type inference algorithm were added to Ruby. Their type system supports object types, since methods are defined inside of objects. Their inference algorithm is constraint-based and applies a set of rewrite rules. They disallow dynamic features of Ruby, like runtime method redefinition. In our work we did not need to disallow any feature of Dylan, because only non-sealed methods might be replaced at runtime. Their algorithm produces some false type errors because of a union types where a part of the union has already been checked by a conditional, elimination of these type errors is left as future work.

Sage Another approach is hybrid type checking [10], which contains a Dynamic type for the unknown values. The type system supports type refinements, which are constraints for types, similar to limited types in Dylan, but more powerful. Additionally types are first-class values (types can be returned from functions, and it can be abstracted over types). When an expression has a type but another type is expected, and subtyping cannot be computed at compile time, a downcast (runtime type check) is inserted. Also, type checking is deferred to runtime if the subtype algorithm takes too much time. A notable difference to our approach is that refinement types may contain any predicate, which are solved with a theorem prover.

Success typing A similar approach, to use type inference for error detection, has been accomplished in success typing [17]. Success typing was developed for Erlang and is done after the compiler inserted dynamic type checks where needed. The inference algorithm finds code locations where types are definitively disjoint and presents these to the user. The goal was to provide documentation and error messages at compile time of unmodified Erlang code.

Multiple dispatch The selection of a method based on all argument types, not only the first, is called multiple dispatch. An empirical study of multiple dispatch was done in [22].

Formalization of type safety and multiple dispatch was done in Cecil [7, 16]. In contrast to our work, Cecil has a prototype-based object system while Dylan has a class-based object system.

Kea [21] is a statically typed programing language which integrated multi-methods and polymorphism. In Kea methods may still be organised within classes to preserve encapsulation. A result of this is that the first argument is still given a special status.

Parametrized types Integration of parametrized types into GOO, a dynamically typed programming language focussing on real time audio and video transformation, was done in [15]. This is a generalization of Dylan's limited types. It contains ideas how to cope with parametric polymorphism and generic functions, which served as a base for the described interaction in that paper. In GOO function types cannot be parametrized, which is a major effort of this paper.

6. Conclusion and further work

In this paper we integrated gradual typing into Dylan, a language supporting multiple dispatch, and additionally added union-based subtyping, multiple and optional parameters and multiple return values. It uses the optional type annotations of Dylan as a base of the type inference. The extensions to the Dylan syntax (to support type variables and function types) are backwards-compatible, so existing Dylan code can be used with the new type inference.

The type inference algorithm was implemented in the mainstream Dylan compiler. The results look promising: the type inference algorithm is more precise and enables more optimizations compared to the original implemented algorithm [2]. The described algorithm reports more type errors at compile time and the generated code contains fewer computations which may fail at runtime, namely type assertions and generic function calls. Additionally we implemented function types and parametric polymorphism in Dylan, which allows developers to specify more expressive types.

The running speed of the implemented type inference algorithm is currently slower than of the original algorithm. This has two main causes: on the one hand it is not incremental, on the other hand the control flow graph is first converted into a static single assignment form. Additionally, in order to improve performance, each polymorphic method which has been upgraded to a monomorphic method should be cached.

An animated graph visualization[4] was developed and will educate people who are interested in compiler construction, especially in compiler optimizations and type theory to get a deep understanding of what a compiler does during optimization and type inference.

Further work will include a formalization and proofs of the developed type system and inference algorithm. Most of the rules are obtained from standard literature [23] and the gradual typing system [26–28], but there are no formal proofs of the combined type system.

The presented treatment of subtyping in the type inference algorithm is inadequate. There has been done some work by Pottier [24] which uses positive and negative annotations on data flow nodes.

Dylan's list comprehension is defined with variable-arity arguments, a recently published paper covering variable-arity polymorphism [30] will be integrated.

Neither collections nor hash tables currently make use of parametric polymorphism. Every hash table lookup, store and rehashing involves a generic function call. The collection implementation has a lot of duplicated code, using copy-down methods, for performance when specific element types are stored in the collection.

An additional feature for Dylan's type system would be occurrence typing [32], which uses information of introspection functions. The method `instance?(<object>, <type>) => (<boolean>)` if used in a test of a conditional can be used to type the object in the consequence of the conditional to be of the specified type. This has been formalized in [32] and additionally composed predicates can be used to gather more precise type information.

Acknowledgments

Thanks to Dirk Kleeblatt, who supervised the author's diploma thesis at TU Berlin, for his valuable feedback and invested time. Also thanks to Peter Sestoft, the author's PhD supervisor at IT University of Copenhagen, for valuable feedback on this paper. And thanks to

[4] http://visualization.dylan-user.org/

Peter Housel for his feedback and wording improvements. Finally thanks to the anonymous reviewers for their valuable feedback.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. 1992.

[2] O. Agesen. Concrete type inference: Delivering object-oriented applications. Technical report, 1996.

[3] J. Bachrach and K. Playford. D-expressions: Lisp power, Dylan style. 2001.

[4] K. Barrett, B. Cassels, P. Haahr, D. Moon, K. Playford, and P. Withington. A monotonic superclass linearization for Dylan. *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Oct 1996.

[5] G. Bracha and D. Griswold. Strongtalk: typechecking Smalltalk in a production environment. *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, Oct 1993.

[6] R. Cartwright and M. Fagan. Soft typing. *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, Jun 1991.

[7] C. Chambers. Object-oriented multi-methods in Cecil. *ECOOP'92: Proceedings of the European Conference of Object-Oriented Programming*, pages 33–56, 1992.

[8] M. Furr, J. hoon An, J. Foster, and M. Hicks. Static type inference for Ruby. *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, Mar 2009.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns. *Book*, 1995.

[10] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. *Scheme and Functional Programming Workshop*, 2006.

[11] F. Henglein. Dynamic typing. *ESOP Proceedings of the 4th European Symposium on Programming*, 582:233–253, Apr 1992.

[12] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, Oct 1995.

[13] G. Huet. Resolution d'equations dans les langages d'ordre 1, 2, ..., omega. 1976.

[14] E. Kidd. Efficient compression of generic function dispatch tables. *Darthmouth College Computer Science Technical Report TR2001-404*, pages 1–22, Nov 2001.

[15] J. Knight. Parametrized types for GOO. *Master thesis*, 2002.

[16] G. T. Leavens and C. Chambers. Typechecking and modules for multi-methods. *ACM Transactions on Programming Languages and Systems*, 17:1–15, 1994.

[17] T. Lindahl and K. Sagonas. Practical type inference based on success typings. *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, Jul 2006.

[18] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. *ECOOP'08 Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 260–284, 2008.

[19] T. Mann and J. Bachrach. Harlequin Dylan runtime system. 1995.

[20] H. Mehnert. Extending Dylan's type system for better type inference and error detection. *Diploma thesis*, pages 1–92, Oct 2009.

[21] W. B. Mugbridge, J. G. Hosking, and J. Hamer. Functional extensions to an object-oriented programming language. Technical report, 1990.

[22] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple dispatch in practice. *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, Oct 2008.

[23] B. C. Pierce. *Types and Programming Languages*. 2002.

[24] F. Pottier. A framework for type inference with subtyping. *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, Jan 1999.

[25] A. Shalit, D. Moon, and O. Starbuck. *The Dylan Reference Manual*. Aug 1996.

[26] J. G. Siek and W. Taha. Gradual typing for functional languages. *Scheme and Functional Programming 2006*, 2006.

[27] J. G. Siek and W. Taha. Gradual typing for objects. *European Conference of Object-Oriented Programming 2007*, 2007.

[28] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. *Dynamic Languages Symposium*, 2008.

[29] J. G. Siek and P. Wadler. Threesomes, with and without blame. *Proceedings for the 1st workshop on Script to Program Evolution*, pages 34–46, 2009.

[30] T. S. Strickland, S. Tobin-Hochstadt, and M. Felleisen. Practical variable-arity polymorphism. *European Symposium on Programming*, pages 32–46, Feb 2009.

[31] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, Oct 2006.

[32] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed Scheme. *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*, page 395, 2008.

[33] P. Wadler. The expression problem. 1998.

[34] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. *ESOP'09 Proceedings of the European Symposium on Programming*, pages 1–16, 2009.