Diploma Thesis

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

Hannes Mehnert
hannes@mehnert.org

Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik
Übersetzerbau und Programmiersprachen

October 1, 2009

Die selbständige und eigenhändige Ausfertigung versichert an Eides statt

| ______________________ | ______________________________________ |
| --- | --- |
| Ort, Datum | Hannes Mehnert |

# Abstract

There are dynamically and statically typed programming languages. Both have different features, statically typed contain early (type) error detection and optimization possibilities due to type inference. Dynamically typed contain dynamism and enable rapid prototyping. This thesis is about integration of static features into a dynamic programming language.

Dynamically typed programming languages rarely have a formal type system, and thus cannot be reasoned about. The correctness of type checking and type inference algorithms is seldom proven.

While there are already some theoretical observations, emphasis is put on visualiziation and verification of results immediately. Due to the choice of using a real programming language the possibility to estimate practicability of integration of the type system into existing software. A complete chapter is dedicated to the practical implementation of the type system.

The focus group of the thesis are researchers and developers which are interested in type systems and dynamic programming languages. Programming language and compiler developers of other programming language than Dylan can benefit from lessons learned. Due to the formal basis bugs in the Dylan compilers were found, analogous implementations can benefit from the results of this thesis and enhance the software quality of other programming languages.

Developers get tools which allow a more precise static analysis (for error detection and optimization) during compilation.

During the research a visualization of the control and data flow was developed as a powerful tool to comprehend the workflow of the compiler and to identify and avoid bugs during compilation. The visualization helps to understand why the compiler is not able to track type correctness of certain expressions and why the compiler cannot optimize certain expressions. This will also help other developers of Dylan, as well as other programming language researches to get an understanding of certain optimizations.

Besides the visualization of the control and data flow, also the type inference algorithm is visualized. This enables future developers to find bugs more easily and to track misunderstandings between developer and compiler.

Finally the practical benefit of the implemented features is examined. The metrics used are the number of generic function calls and type checks at run time. Bugs found with the new type system and type inference algorithm in the standard Dylan library and in the Dylan compiler are presented.

# Contents

**1 Introduction** **1**

**2 Overview of Dylan and the Compiler** **3**
2.1 Dylan . . . . . 4
2.2 DylanWorks . . . . . 6
2.3 Parser and Lexer . . . . . 7
2.4 Enhancing with Semantics: Definitions . . . . . 7
2.5 Model Objects . . . . . 8
2.6 Flow graph . . . . . 10
2.6.1 Data Flow . . . . . 10
2.6.2 Control Flow . . . . . 11
2.7 Conversion to Flow Graph . . . . . 21
2.8 Type Inference . . . . . 21
2.8.1 Drawbacks . . . . . 22
2.9 Optimization . . . . . 22
2.9.1 Dispatch Upgrading . . . . . 23
2.9.2 Optimizer Flow . . . . . 26
2.9.3 Example . . . . . 26

**3 Gradual Typing** **29**
3.1 Design Considerations of Unification Based Inference . . . . . 31
3.1.1 Dynamic Types as Type Variables . . . . . 31
3.1.2 Ignore Dynamic Types During Unification . . . . . 32
3.1.3 Well-typed After Substitution . . . . . 32
3.2 Type System for $\lambda^{?\alpha}_{\rightarrow}$ . . . . . 32
3.3 An Inference Algorithm for $\lambda^{?\alpha}_{\rightarrow}$ . . . . . 33
3.3.1 Constraint Generation . . . . . 34
3.3.2 Constraint Solver . . . . . 34
3.4 Subtyping . . . . . 41

**4 Type System for Dylan** **43**
4.1 Type Constructors . . . . . 43
4.2 Relations on Types . . . . . 44
4.2.1 Subtype . . . . . 45
4.2.2 Disjointness . . . . . 49
4.3 Extensions to Type Inference Algorithm . . . . . 50
4.3.1 Constraint Generation . . . . . 50
4.3.2 Solve Algorithm . . . . . 55

4.4 Interaction of Parametric Polymorphism and Generic Functions . 56

**5 Changes to the Compiler and Results 59**

5.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.2 Enhancing with Semantics: Definitions . . . . . . . . . . . . . . . 59
5.3 Model Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
5.4 Conversion to Flow Graph . . . . . . . . . . . . . . . . . . . . . . 60
5.5 Static Single Assignment . . . . . . . . . . . . . . . . . . . . . . . 60
5.6 Compiler Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
5.7 Type Inference . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.7.1 Type Graph . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.7.2 Type Environment . . . . . . . . . . . . . . . . . . . . . . 66
5.7.3 Signature Upgrading . . . . . . . . . . . . . . . . . . . . . 67
5.7.4 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.8 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.8.1 Polymorphic Functions . . . . . . . . . . . . . . . . . . . . 71
5.8.2 Comparison to Original Type Inference . . . . . . . . . . . 72
5.8.3 Problems Encountered in DylanWorks and the Dylan Library . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.8.4 Comparison to Polymorphic Dylan Library . . . . . . . . . 73

**6 Related Work 77**

**7 Conclusion and Further Work 79**

**A Zusammenfassung 85**

# Chapter 1

# Introduction

Programming languages can be separated into statically and dynamically typed languages. Statically typed programming languages check types at compile time while dynamically typed check at run time. Some statically typed programming languages like Haskell have a formalized type system, based on the $\lambda$ calculus. A formal type system enables to proof that a program is well typed and thus cannot fail because of type errors. Type checking algorithms normally have preservation and termination proofs. Type inference algorithm have soundness and completeness proofs. Untyped languages like Scheme or Ruby have no formalization of their type system at all.

Dynamic typing enables fast adaption of changing requirements and supports information, like user input or network data, whose value and even type is unknown at compile time. Static typing provides type error detection at compile time, yields more performance and enables formal reasoning of software.

There are numerous ways to combine static and dynamic typing. On the one side, several dynamic languages support optional type annotations to increase performance, but not the amount of static type checking. An example is Common LISP. Similarly soft typing [CF91] infers types to improve performance, but not for static error detection. On the other side, statically typed languages like Haskell introduce a `Dynamic` type and `typecase` to become more flexible. These languages do not encourage programming in a dynamic style, because the programmer has to insert explicit casts to and from the `Dynamic` type. Deployed statically typed languages like Java, C++ and .NET also integrate introspection, which needs run time type information, a feature originated from dynamic languages.

The thesis will show whether different concepts of dynamic programming languages can be integrated into a formal type system easily, thereby diminishing the friction between static and dynamic typing. Hopefully in the future developers will be able to rapidly prototype an application in an untyped language, and once finished, adding type annotations to the parts of the application which should be checked statically. Thus, the optional type annotations will be used for static type checking (error reporting) and type inference (performance). The programming language researched is Dylan. Additionally the type syntax of Dylan is extended by parametric polymorphism and function types to allow more narrow type specifications. For these extensions a motivating code example is `map(method(x) x + 1 end, #(1, 2, 3))`, which applies `x + 1` to

every element of the literal list `#(1, 2, 3)`. Due to the lack of expressiveness of Dylan types, the method `map` is typed as `(<function>, <collection>) => <collection>`. There is no dependency between the function signature and the collection elements.

This thesis is based on current research in type theory. To understand the constraints and ensure a basic understanding of syntax, a short introduction into the programming language Dylan and of the extended compiler, is essential. The main working steps of the compiler and the starting points of the integration of the type system are presented in chapter 2.

Subsequently this thesis introduces in chapter 3 the used theoretical basics of the type system named gradual typing, which is deduced from the simple typed $\lambda$ calculus. Afterwards extensions in form of subtyping and a type inference algorithm is presented, to be able to use the type system for Dylan.

After the theoretical overview this paper applies the results of my research to a concrete type system for Dylan in chapter 4. During the application I extend Dylan with function types. To specify functions with more narrow types, I extended Dylan also with parametric polymorphism. The researched type inference algorithm conveniently supports parametric polymorphism. The type inference algorithm is extended to be applicable for Dylan.

After giving an overview of this concrete type system, an existing Dylan compiler is extended with this type inference algorithm and the other mentioned type feature in chapter 5. This enables me to verify the assumptions and is now available for the programming language researchers to further work on it.

During my research I developed a visualization of the control and data flow as a powerful tool to comprehend the workflow of the compiler and to identify and avoid bugs during compilation. The visualization helps to understand why the compiler is not able to track type correctness of certain expressions and why the compiler cannot optimize certain expressions. This will also help other developers of Dylan, as well as other programming language researches to get an understanding of certain optimizations. Besides visualization of the control and data flow now the steps and state of the type inference algorithm are visualized. This enables future developers to detect errors more easily and to understand bugs in the compiler.

In chapter 5 I finish by investigating the practical gain achieved by implementing the mentioned features. The metrics used are the performance, a measurement of the number of generic function calls and type checks at run time. I also present bugs in the Dylan standard library as well as in the Dylan compiler, which were found by the implemented type inference algorithm.

In chapter 6 related work is discussed. In the final chapter 7 I conclude and mention further work.

# Chapter 2

# Overview of Dylan and the Compiler

Dylan is an object-oriented dynamically typed programming language. Its design was mainly influenced by LISP, the Common Lisp Object System and Smalltalk. In contrast to LISP Dylan has an ALGOL like syntax. In 1996 the Dylan Reference Manual [Sha96] was published, serving as the language standard. Classes and functions are first class objects and share the same namespace. Generic functions and multiple dispatch are available. Multiple inheritance, ad-hoc and inclusion polymorphism (as defined by [CW85]) are supported, as well as optional keyword arguments, variable arity arguments, macros (meta-programming), anonymous methods, closures and higher order functions. Dylan was developed by Apple Cambridge, Harlequin and Carnegie Mellon University.

The compiler extended in this paper is Open Dylan, formerly named DylanWorks and Functional Developer. It was developed at Harlequin, later Functional Objects, and is open source (Dual-License: LGPL and Functional Objects Library Public License Version 1.0) since March 2004.

DylanWorks is written in Dylan, it was bootstrapped by an emulator written in LISP. It compiles Dylan code into an intermediate representation named Dylan Flow Machine Language (DFM). This has currently several running backends: one generates C code, another is Harlequin Abstract RISC Processor (HARP), which is a virtual processor with a RISC instruction set. HARP has a native x86 backend as well as an incomplete PowerPC backend. There is also a debug backend for DFM, which pretty prints control and data flow elements. Another incomplete backend for DFM targets the JVM. The extensions of the Dylan compiler introduced do not require changes to these backends, apart from minor fixes.

DylanWorks has two operational modes, one being used for production code and the other being used for development mode. The production mode applies more optimizations to the code. In development mode definitions can be interactively compiled and inserted into the running program (hot-code update), debugging is easier, because functions are not aggressively inlined, so stack frames for every call are available, breakpoints can be set everywhere, and a read-eval-print-loop (REPL) is provided. This thesis focuses on production

mode, but is also applicable to development mode.

In the next section, a detailed overview of Dylan will be given. Afterwards DylanWorks workflow is introduced.

## 2.1 Dylan

The Dylan language provides generic functions and multiple dispatch, just as Common Lisp Object System and Smalltalk do. In contrast to most object oriented programming languages, methods are not written inside of a class. Methods are defined in the top level namespace and do not contain an implicit this or self argument. All arguments have to be named explicitly and their type can be specified (defaults to the top of the type hierarchy). All methods with the same name are required to have the same number of required arguments and they construct a generic function with this name. At a call site, all methods of the generic function are taken into account and the most specific with regard to all actual argument types is executed. This is described in more detail in section 2.9.1. In contrast to common object oriented programming languages, the method lookup is not done in the class of the first argument type (single dispatch), but takes all arguments into account (multiple dispatch). Multiple dispatch has also been studied in Cecil [Cha92]. Cecil has a classless (prototype-based) object model, whereas Dylan has class-based object model.

Each generic function may have any number of required arguments, optional keyword arguments and variable arity arguments. A method of a generic function must accept the same number of required arguments. If a generic function is specified explicitly, each methods argument types must be subtypes of the specified generic argument types. If a generic is not specified explicitly, it defaults to the top of the type hierarchy in all argument positions. It is also possible to define a generic function with a single method by using `define function` instead of `define method`.

Classes may inherit from multiple other classes, where the topmost class is called `<object>`. The member variables of a class are called slots in Dylan. A class inherits all slots of all superclasses, without duplicating slots if the superclass graph contains a class more than once. Access to these slots is managed via function calls; for each slot the compiler generates a getter and a setter function. If the slot is marked read only (by the keyword constant), only a getter function is generated, and the value cannot be changed after instantiation. The main purpose of classes is to structure and encapsulate data, not to encapsulate code, since methods are defined outside of classes.

Additionally a concept called sealing is available which prevents further extension of a definition, either a generic function or a class. Sealing is similar to `final` in Java. Sealed definitions enable some optimization opportunities, because they are completely known at compile time and can be reasoned about. A huge optimization opportunity is upgrading generic function calls to method calls at compile time, which is described in detail in section 2.9.1.

A definition's scope is its namespace, called module in Dylan. A module imports bindings from other modules and exports bindings. During import a subset of bindings can be selected and the imported bindings can be renamed. Access to slots can be controlled by exporting or not exporting getter and setter methods (equivalent to `public, private, protected` in other languages).

```
block(return)
  for (index from 0)
    if (list[index] == search)
      return(index)
    end
  end
end
```

Listing 2.1: Example of a protected block with fast return

Modules are grouped in libraries, which are the unit of compilation. Libraries export modules and import other libraries or parts thereof.

Dylan knows no explicit return statement, expressions return the value of its last statement. It also supports multiple return values; the developer does not need to construct a collection and return this. Return values not bound at call site are thrown away. An example is the `match` method of the regular expression library, which receives a string and a regular expression as arguments and returns a boolean value, whether the regular expression matched, and the matching positions. This provides flexibility on the call site: if the caller is only interested whether the regular expression matched, only the first return value is bound; if the caller is interested in the first match, also the second return value is bound; if the caller is interested in all matches, all return values can be bound.

Dylan supports LISP-like condition handling. Handlers can be installed and if a condition occurs, the most recent matching handler is executed. This can call the next handler in the chain. In most programming languages first the stack is unwound up to the context of the exception handler before calling the handler. In Dylan instead first the handler is executed, enabling the handler to modify the stack and continue the program's execution, restarting the last call with modified variables. This also allow to interactively debug the current state, in contrast to other programming languages, where the condition has a stack trace attached, which can be printed, but not debugged.

In order to gracefully develop with conditions, protected blocks are available. These blocks may contain a named exit function, a cleanup clause for resource deallocation and exception clauses which setup exception handlers. An example of a named exit function (here named `return`) is given in listing 2.1, where the index of the first match is returned, without iterating over the rest of the list.

An example which closes the file, even if a condition is thrown during processing, is given in listing 2.2. Due to the usefulness of this code fragment, it is wrapped in a macro called `with-open-file`. This macro receives a filename and a body which processes the file stream. By using this macro, a developer does not need to manually close the given file.

Dylan supports metaprogramming by pattern matching macros. The syntax of Dylan can be extended by macros easily; in fact most of syntactic sugar is internally implemented by macros. The macro system is not as powerful as the one of LISP, namely procedural macros are missing. These procedural macros are not trivial to implement without eval and without a s-expression syntax.

The syntax of Dylan is ALGOL like. By convention, classes are surrounded

```
block()
  let file-stream = open-file(filename);
  process-file(file-stream)
cleanup
  close-file(file-stream)
end
```

Listing 2.2: Example of resource deallocation with a protected block

```
define class <point> (<object>)
  slot x :: <number>, init-keyword: x:;
  slot y :: <number>, init-keyword: y:;
end
```

Listing 2.3: `class <point>`

with angle brackets (`<object>`). There are other coding conventions, for example a method returning a boolean value should contain a question mark as last character in its name. There are only few reserved words in the Dylan grammar [Sha96, Appendix A]. Names can include "special" characters like ?, -, +, etc. Thus, method names like `any?` and `add!` can be used.

A simple class definition is shown in listing 2.3, which defines the class `<point>` with the single superclass `<object>`. Each point has two slots, `x` and `y`, each of type number. Each slot can be initialized at instantiation time with the specified init-keyword, `x:` and `y:`, which are optional keyword arguments to the instance creation method `make`. A simple method definition is shown in listing 2.4, which defines a method named `double`. It has the single argument `a` of type integer and a single integer return value, named `result` only for documentation purposes. It returns its argument multiplied by 2.

In contrast to LISP Dylan has a clear separation between compile time and run time, a Dylan compiler is not needed at run time. While this facilitates small and fast binaries, it contracepts some dynamism like the meta object protocol and procedural macros.

## 2.2 DylanWorks

In the subsequent section several libraries of the DylanWorks compiler are described, because this thesis extends most of them. The control flow of the compiler is shown in figure 2.1. For better understanding the method `double`

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

Listing 2.4: method `double`

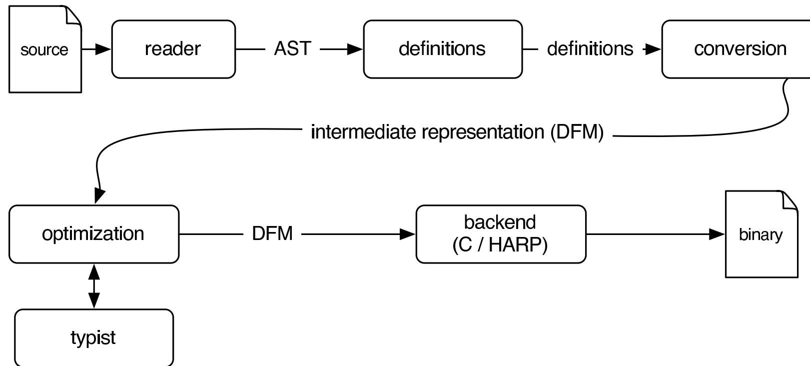


Figure 2.1: Overview of the Open Dylan compilation process

(listing 2.4) will serve as an example through this section.

## 2.3 Parser and Lexer

The parser and lexer reads the Dylan source from a stream, tokenizes it, annotates source location information, and builds an abstract syntax tree. The parser is automatically generated from the grammar in LALR(1) form, specified in [Sha96, Appendix A]. The parser does basic consistency checks like insularity of blocks and balance of parentheses.

From listing 2.4 Dylan's parser generates the fragment tree shown in figure 2.2 which consists of a body definition fragment capturing the information that a method is defined. It contains seven children: the name of the method, `double`, a parented argument list with the single argument and its type, the token `=>` which separates arguments and values, a parented values list (again a single value with a type) and finally the body. The body consists of the fragments `a`, the binary operator `*` and `2`.

## 2.4 Enhancing with Semantics: Definitions

Once the abstract syntax tree is built, semantic information is incorporated. Macro expansion takes place, as well as a translation into definition objects, which model the semantics of Dylan. Different definitions, like method definitions, class definitions, etc. instantiate different definition classes. The transformation to definition objects is implemented by pattern matching on the abstract syntax tree.

The definitions library generates from the syntax tree (figure 2.2) of listing 2.4 the definition object shown in figure 2.3. The single object is a method definition, consisting of a signature specification and a body. The signature specification contains an argument and a value list, both contain a single element. These elements each consist of a name (`a`, `res`) and a type expression (both `<integer>`), all former fragments of the syntax tree. The body has a single constituent, a binary operator call fragment. This binary operator call consists of the function fragment `*` and the argument fragments `a` and `2`, also

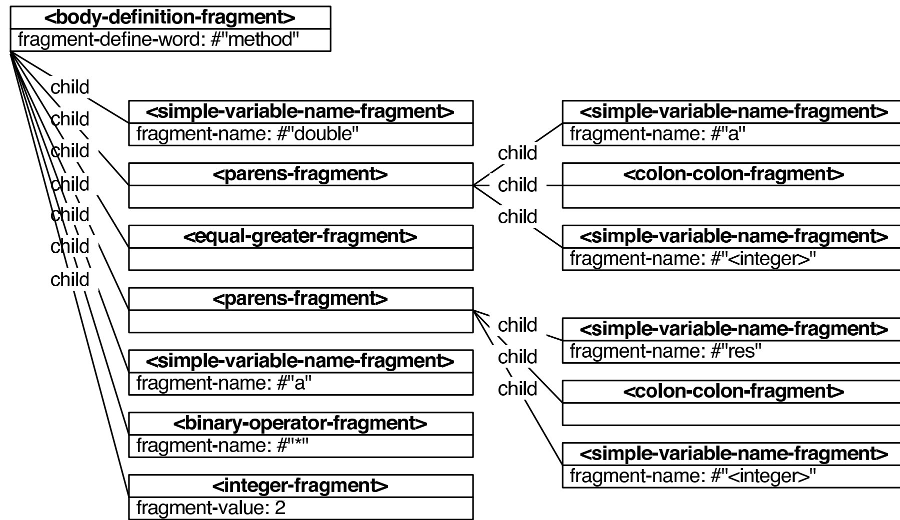


Figure 2.2: UML object diagram of parsed method `double`

```
define side-effect-free stateless dynamic-extent
 &primitive primitive-vector-element
  (x :: <simple-object-vector>, index :: <raw-integer>)
  => (value :: <object>)
```

Listing 2.5: `primitive vector-element`

former fragments of the syntax tree.

## 2.5 Model Objects

The library `dfmc-modeling` declares primitive operations, types and relations on types. It is used in the flow graph, conversion and optimization libraries and in the backends. A model object has a representation both at compile time and run time, thus it can be reasoned about at compile time or passed to a backend and be used at run time. Every compiled Dylan library consists of a binary, as well as of metadata, stored in a separate file, using the Dylan object oriented database. This metadata includes the flow graph, which is needed for optimizations like inlining, and the signature, which is needed to testify type safety of cross library calls.

The declared primitives have to be implemented in the targeted backend. Listing 2.5 shows the primitive `vector-element`, which has two arguments, a `<simple-object-vector>` and a `<raw-integer>` used as index, and returns the object at the given position. Annotations of primitives provide additional information to the compiler: `side-effecting`, whether the primitive raises side effects or not, `stateful`, whether it mutates state, and `dynamic-extent`, whether arguments are passed to another function. Those properties are used by the compiler to figure out whether a call to a primitive can be constant folded, etc.

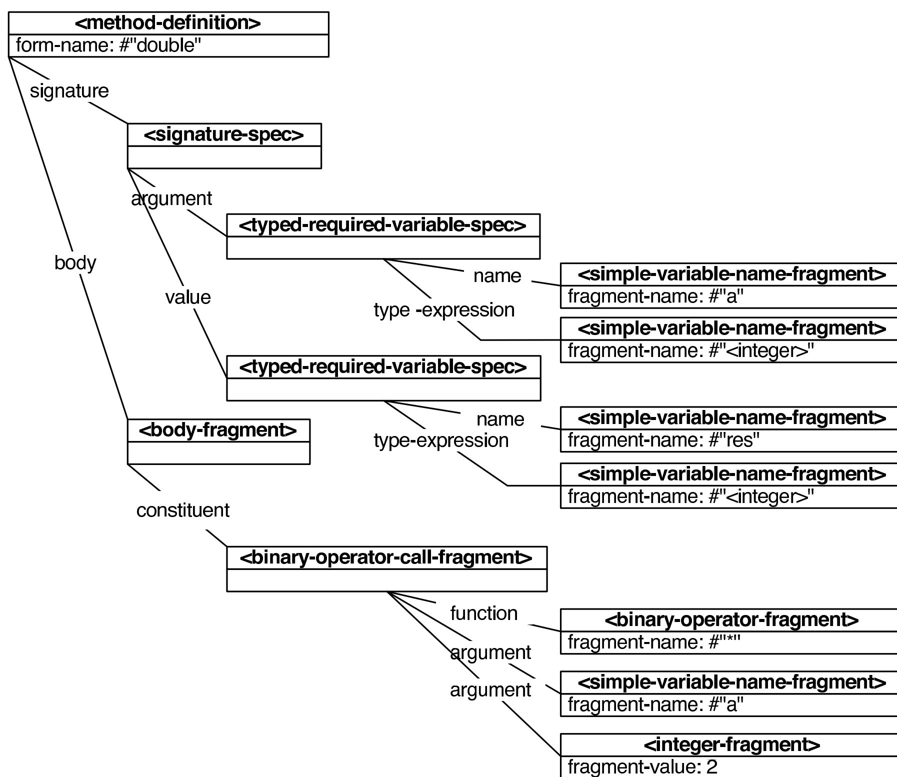


Figure 2.3: UML object diagram of definitions of `method double`

```
define abstract dood-class <referenced-object>
 (<object>)
  slot users :: <list> = #(), init-keyword: users:
end
```

Listing 2.6: class referenced-object

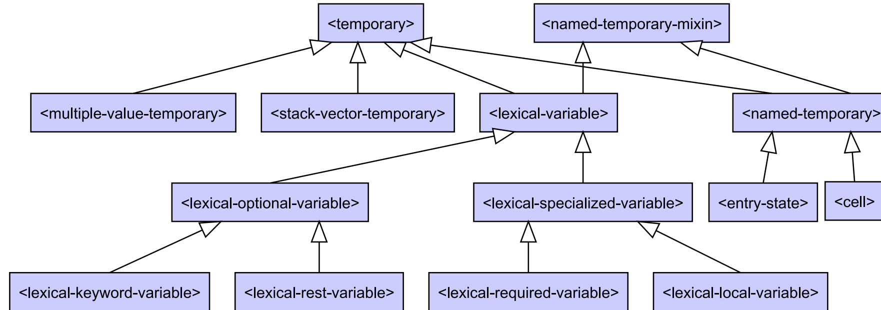


Figure 2.4: Class hierarchy of temporary

## 2.6 Flow graph

The intermediate representation of the compiler is the flow graph. Optimizations and type inference is applied on the flow graph of each top level definition. The flow graph library `dfmc-flow-graph` consists of control flow nodes, subclasses of `<computation>`, data flow nodes, subclasses of `<referenced-object>`, and general utilities for constructing graphs, deleting (sub-)graphs, walking graphs, copying (sub-)graphs, etc. For every top level definition (method, constant, class constructor) a flow graph is generated during the conversion phase (section 2.7). Every flow graph thus embodies a top level definition in the source code, either raw written down code or generated code from syntactic sugar.

### 2.6.1 Data Flow

Every data flow node is an instance of a subclass of `<referenced-object>`, shown in listing 2.6. The adjective `abstract` means that the class is not instantiable. The macro `dood-class` defines a Dylan class and generates support code for marshalling and unmarshalling. The class definition contains a single slot, `users`, which has a type of `<list>` and a default value of the empty list. This slot records control flow nodes which use this data flow node.

There are several noteworthy concrete data flow classes:
`<defined-constant-reference>` which references a constant,
`<method-reference>` which references a `<method>` object, and `<temporary>`.

A `<temporary>` is a value continuation, it does not contain a value, but passes the generating computation's result value to its users. Each `<temporary>` contains additional slots: the computation which generated it, its lexical environment and some properties like liveliness information, whether the temporary is passed to a function call.

There are several subclasses of the temporary class, shown in figure 2.4. Temporaries which contain a name inherit from `<named-temporary-mixin>`, like lexical variables. These lexical variables can also contain a type specializer, represent an argument of a method, either a required or an optional one, or be a local variable. Variables which are assigned, and not only bound once, are represented by instances of the `<cell>` class.

To provide support for multiple return values, the class `<multiple-value-temporary>` represents a variable number of values, either known at compile time to be a specific fixed amount or of variable arity, or completely unknown at compile time.

### 2.6.2 Control Flow

The intermediate representation is based on the $\lambda$ calculus, binding, application and abstraction are present, while there are some additions like conditionals, loops and multiple value support, including variable arity.

Each control flow node is a subclass of the `<computation>` class. Each computation contains a reference to its previous and next computation, thus the control flow graph is a doubly-linked list, exceptions are `if`, `loop` and blocks, containing additional control flow edges.

Every computation generates a data flow node, called temporary, and every computation has a reference to its lexical environment.

The computation class hierarchy is shown in figure 2.5. Some computations are collected into group nodes (for example `call`). They are collapsed in this figure, but will be expanded in further figures.

**Boundary** Every flow graph has a dedicated beginning and end node. The beginning is named `<bind>`, the final node is `<return>`. They cannot be optimized and do not generate any data flow element. The data flow element consumed by the final node is the return value of the top level definition.

**Bindings** A binding is a special computation that creates a data flow node representing an assignment to a real or temporary variable. There are several computations whose main task is to introduce a binding, for example a reference to a global variable creates a temporary containing the value of that variable. If the result of the computation can be statically determined at compile time (e.g. a reference to a constant), it can be replaced by a data flow element. Each binding has at least one data flow input and exactly one data flow output edge. The class hierarchy are shown in figures 2.7 and 2.10. Below some concrete binding classes will be described in detail.

**<temporary-transfer>** The class `<temporary-transfer>` is instantiable. The top level statement `let foo = 42` produces the flow graph shown in figure 2.6. Data flow nodes are bright while control flow nodes are dark. Data flow edges are dashed. Control flow edges are solid.

**<check-type-computation>** Another important temporary transfer computation is `<check-type-computation>`. Each type check contains an additional data flow input, the requested type. At run time, if the data value input is not of the requested type, an error is signaled. If at compile time

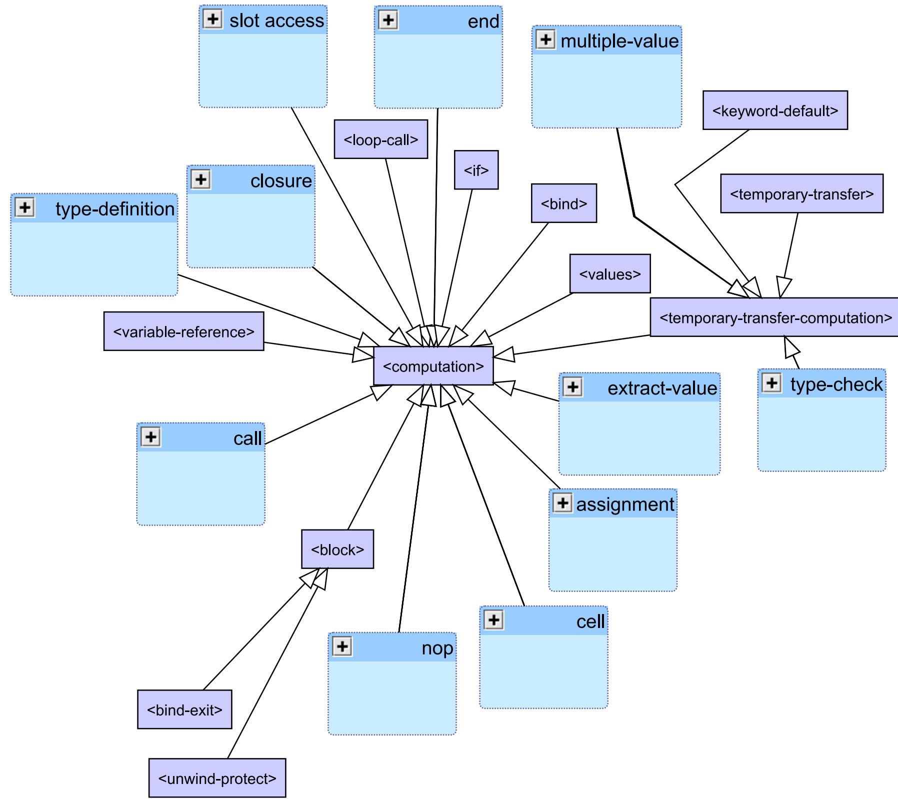


Figure 2.5: Class hierarchy of computations

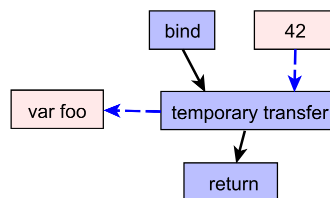


Figure 2.6: Flow graph of `let foo = 42`

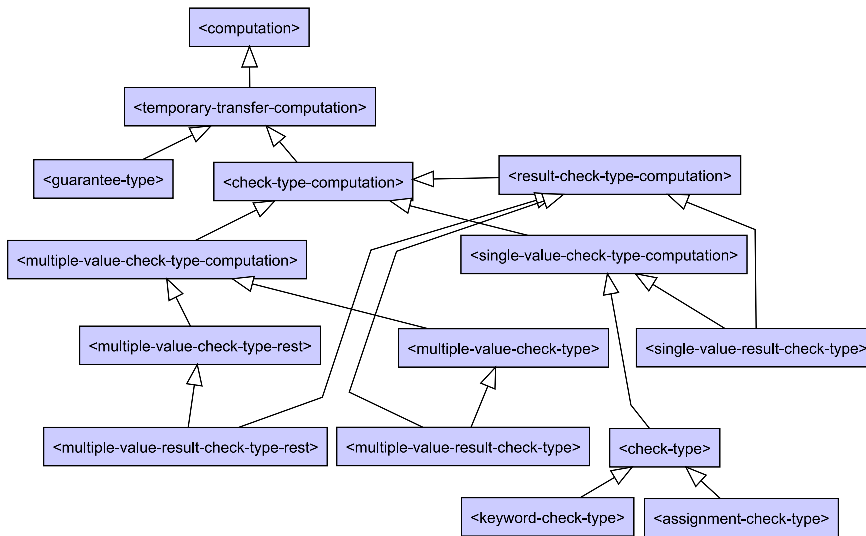


Figure 2.7: Class hierarchy of check type computation

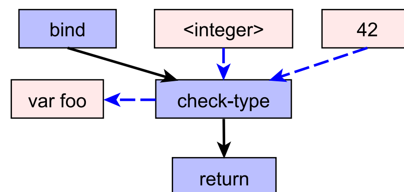


Figure 2.8: Flow graph of `let foo :: <integer> = 42`

the type of the data flow node is known to be a subtype of the requested type, this computation is constant folded and in all succeeding computations the generated temporary is replaced by the data flow input. If both types are known to be disjoint at compile time, a type error is reported at compile time. The complete class hierarchy is shown in figure 2.7. There are distinct classes for single and multiple value temporaries, plus variable arity multiple values. They only differ in the number of types to be checked for (a single type versus a list of types). Another distinction is between "normal" and "result" type checks, each generating different error messages.

The statement `let foo :: <integer> = 42` generates the flow graph shown in figure 2.8

**<guarantee-type>** The computation `<guarantee-type>` allows a developer to inject a type for a binding, which is then used unchecked by the compiler. This is only used at places which are important for performance, where the compiler is not able to guarantee the specific type automatically. Since the type provided is not checked by the compiler, the developer can

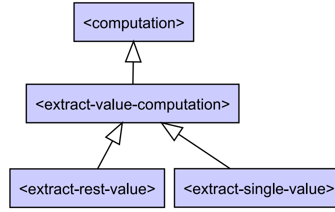


Figure 2.9: Class hierarchy of extract value computations

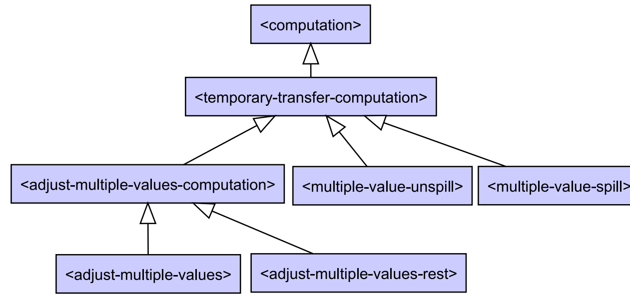


Figure 2.10: Class hierarchy of temporary transfer multiple values

circumvent type safety of the type system by providing a wrong type, thus usage of this computation is unsafe.

**<variable-reference>** A `<variable-reference>` generates a data flow node which contains the value of a global binding.

**Multiple values** There are several operations on multiple-value temporaries: construction, extraction, adjustment, backup and restore. The computation `<values>` constructs a multiple value temporary from multiple single-value temporaries. The computation `<extract-value-computation>` extracts a single temporary from a multiple-value temporary. They are shown in figure 2.9.

Another group of computations adjust a multiple-value temporary to one with more or fewer fixed and variable arity values. Other computations backup (`<multiple-value-spill>`) and restore (`<multiple-value-unspill>`) the multiple value region, which is a distinct thread-local area and thus only one multiple value temporary may be alive per thread at a time. These computations are shown in figure 2.10.

**Application** Application is a major concept in programming languages. In Dylan, function calls are applications. Slot accessors are done by a function call and thus also are applications.

Calling conventions are documented in detail in [MB95]. There are different entry points for every method. An exterior entry point sorts the keyword-value pairs in the canonical order, strips away the keywords, and finally calls

```
define method entry-point
 (a, b, #rest optionals, #key key1, key2);

entry-point(1, 2, key2: 99)
```

Listing 2.7: method entry-point

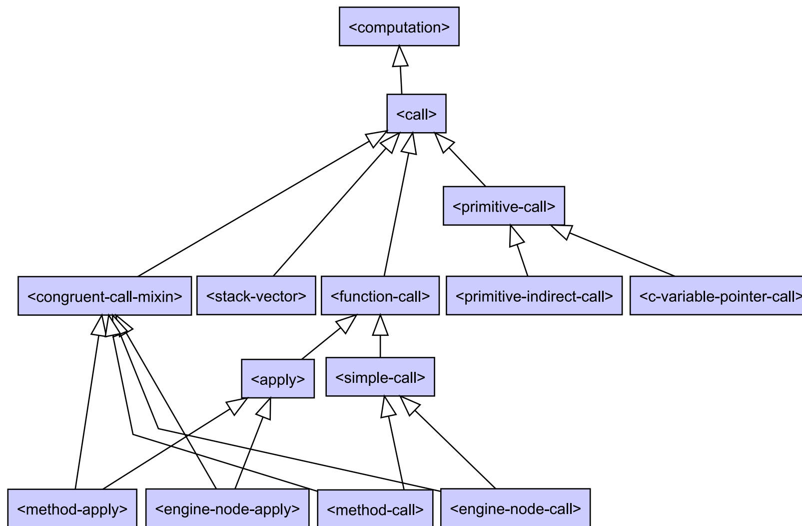


Figure 2.11: Class hierarchy of call computations

the interior entry point as a tail call. An interior entry point is as efficient as possible, and does no work on its own, but just executes the flow graph.

Advantages of this calling convention are that required arguments can always be found at a known offset relative to the stack or frame pointer. Also optional arguments appear in the same order in memory as they would if vectored up as `#rest` parameters. Optional arguments are represented as vectors, so allocating space for them on the stack is trivial.

An example is given in listing 2.7. The exterior entry point of the call gets the parameter list `[1, 2, #"key2", 99]`. The interior entry point gets the parameter list `[1, 2, #f, 99]`, thus keywords have been removed and missingkeywords filled with default values (`#f`).

Figure 2.11 shows the class hierarchy of call computations. A call is congruent if the keyword arguments are in the correct order on the stack and the keywords have been stripped. Therefore congruent calls use the internal entry point.

Each call contains an argument vector of data flow nodes.

**<function-call>** A function call extends a call by providing a reference to a function which is called. This class is abstract.

A simple call is instantiable. The flow graph of the code fragment `let foo = 4 * 2` is shown in figure 2.12.

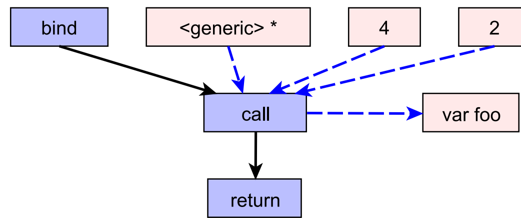


Figure 2.12: Flow graph of `let foo = 4 * 2`

A method call uses the internal entry point.

**<primitive-call>** There are some primitive operations in the compiler, which need to be implemented by the back ends, e.g. arithmetic operations, described in section 2.5. A primitive call contains a reference to a primitive.

**<stack-vector>** A stack vector computation is used to pass variably arity and optional arguments. Any number of non-required arguments are put into a stack vector, passed as the last argument to a method. A stack vector does not have call semantics, but contains an argument list, thus it is a subclass of `<call>` in the class hierarchy.

**Abstraction** The third concept of the $\lambda$ calculus is abstraction. Dylan provides this through method definitions, but also through local and anonymous methods which can capture local bindings (closures). Two computations deal with closures, namely `<make-closure>` and `<initialize-closure>`. Both are direct subclasses of the computation class. The flow graph of `let foo = method(x) x + 1 end` is shown in figure 2.13. While `<make-closure>` allocates space for local variables, `<initialize-closure>` initializes closure data like the lexical environment. The `T` in a data flow node is an unnamed temporary containing a single value, while `MVT` is a multiple value temporary.

**Conditional** Conditionals are used to structure programs. They express behaviour based on a test condition, and execute one of multiple code paths, depending on the outcome of the test associated with the conditional. The computation `<if>` is shown in figure 2.5.

The computation `<if>` has a single data flow input, which is the result of the test. It has two control flow outputs, one consequent and one alternative branch. If the value of the data flow input is true, the consequent branch will be used, otherwise the alternative branch. The next computation reference of an `<if>` is its corresponding `<if-merge>`. An `<if-merge>` consists of two control flow and two data flow inputs. The single data flow output is the value of the branch which was taken. The flow graph of `if (a) 1 else 2 end` is shown in figure 2.14.

**Loop** A loop statement contains a sequence of statements, its body, which is specified once but may be executed several times depending on the context: Either for a pre-specified amount, or for every element of a colletion or until a

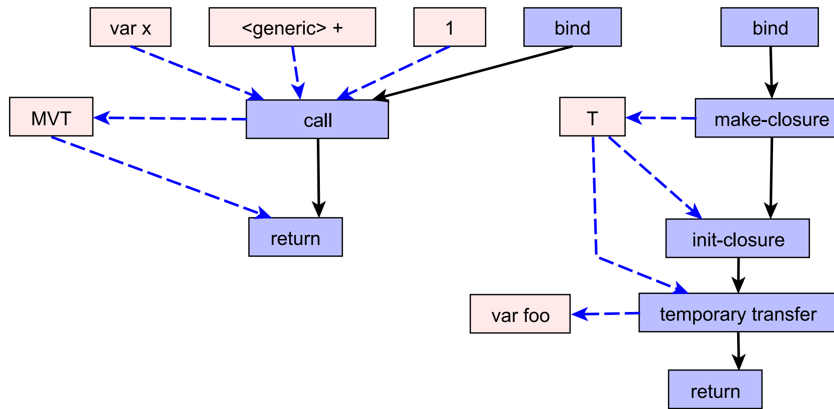


Figure 2.13: Flow graph of `let foo = method(x) x + 1 end`, the closure `x + 1` is shown on the left side. `T` represents an unnamed temporary, `MVT` an unnamed multiple value temporary.

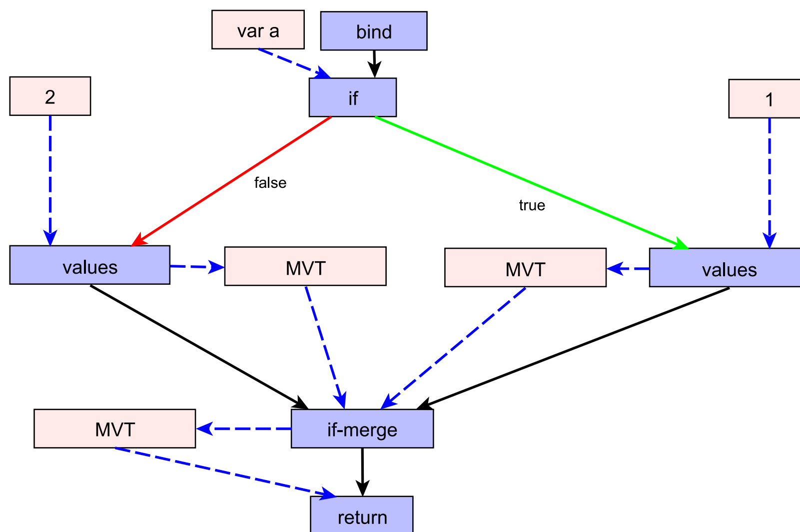


Figure 2.14: Flow graph of `if (a) 1 else 2 end`

condition is met. A loop can be expressed using recursion. Tail recursion is a special case of recursion where the last statement of a defined method is a call to itself. In Dylan these tail calls are optimized to loops, because loops are less expensive than calls. They do not need additional stack frame allocation and parameter passing. A detailed analysis of loop is done in [Shi05].

A basic `<loop>` contains a body, which is a reference to the first computation of the body. Each loop variable value is computed in a `<loop-merge>` computation, which contains two data flow inputs. The `<loop-call>` (continue) returns to the beginning of the loop body. The `<end-loop>` (break) returns from the loop, and proceeds with the next computation after the loop.

The flow graph of `for (i from 0 below 20) i + 2 end` is shown in figure 2.15.

**Blocks** A block contains one or two sequences of computations, which are executed in a specific manner. Its body is always present and represents the sequence of computations executed under normal circumstances. So called unwind-protect frames also contain a second sequence of cleanup computations executed if the body throws a condition. In a bind-exit block a binding is available to proceed execution at the end of the sequence. Blocks can be used for resource cleanup, to handle dynamic bindings of global variables and condition handler chains. The block class inheritance diagram is shown in figure 2.5. Block syntax and semantics were already introduced during the description of Dylan in section 2.1.

**Slots** The binary data representation of classes is composed of slots. Slot access is done via a generic function call. If the arguments are at compile time guaranteed to be a (possibly static) offset into a memory region containing the object, the generic function call is upgraded to a direct slot access. The class hierarchy of slot computations is shown in figure 2.16. Collections are represented as repeated slots, where access to an element is provided by simple pointer arithmethics.

**Assignment** State mutation is abstracted by a concept called assignment. It modifies one or mutliple variable's representation, be it in processor registers or a memory locations. Different assignment classes are shown in figure 2.17. Every assignment that does not mutate a global variable only exists in the initial control flow graph and is converted to a value cell model, as explained later. The `<conditional-update!>` is only used for atomic increment and decrement operations. The `<definition>` computation is used to define global bindings (variables and constants). The `<redefinition>` supports interactive use.

**Cells** Value cell is an abstraction to model assignment. It consists of memory regions which can be mutated. Every variable which has assignments is converted to a cell, represented by the data flow class `<cell>`. The generator of a cell is the computation `<make-cell>`. Every access to the variable is handled by a `<get-cell-value>` computation, which reads the value of the cell and every assignment is converted to a `<set-cell-value!>` computation, which writes the value to the cell. This standard abstraction allows easy handling of assignments during optimization and binary generation, since data dependencies are

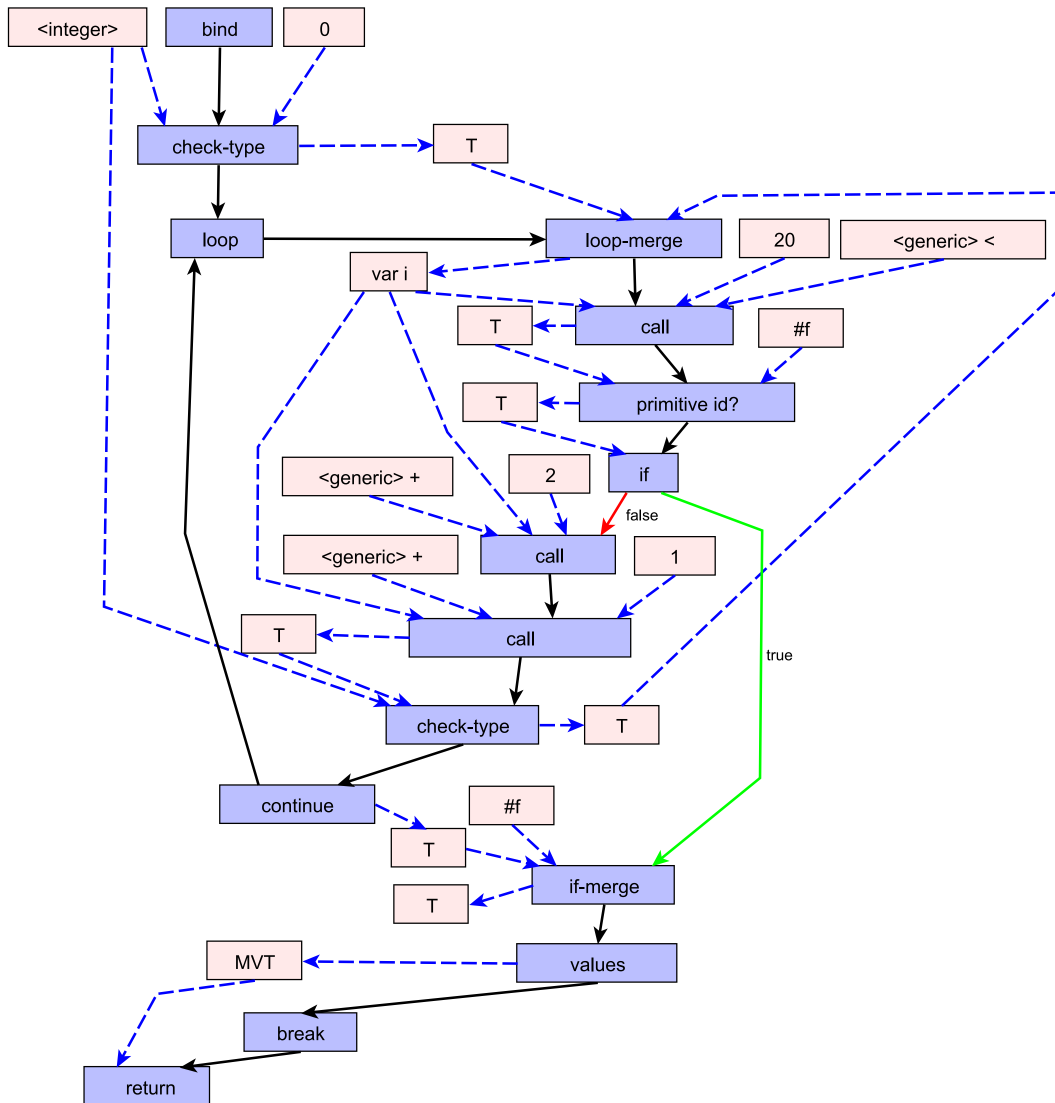


Figure 2.15: Flow graph of `for (i from 0 below 20) i + 2 end`

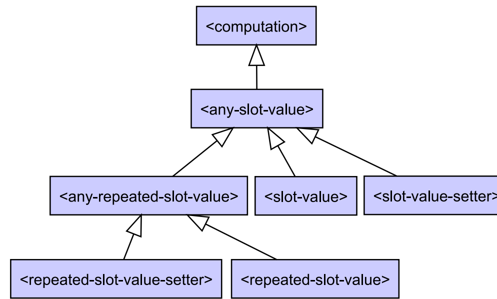


Figure 2.16: Class hierarchy of slot computations

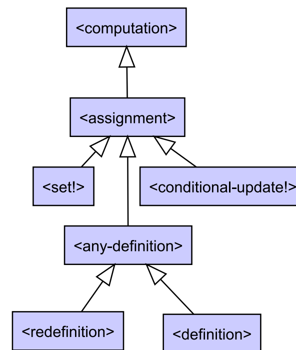


Figure 2.17: Class hierarchy of assignment

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

Listing 2.8: `method double`

in the control flow. All three mentioned cell operations are direct subclasses of the `<computation>` class.

## 2.7 Conversion to Flow Graph

The compiler uses its `conversion` library to convert definition objects from the definitions phase (section 2.4) to model objects (section 2.5), one at a time. This phase includes resolution of referenced bindings, construction of data and control flow graph (section 2.6), setup of the lexical environment and error checking. The error checking of a defined method includes a test whether the generic function may be extended in the specific domain and whether the signature of the method is congruent to the signature of the generic function.

The conversion library also inserts run time type checks where needed: If a binding is introduced which contains a type specializer or a variable gets assigned, a run time type check is inserted. Also if a method has a specification of return values, a check type computation checking for those is emitted. Thus, type annotations serve as contracts (enabling type safety) and are verified by type checks which raise an exception if the value is not of the specified type. If the type inference can at compile time guarantee those checks to succeed, the optimizer removes them later on.

The method `double` serves as an example, again shown in listing 2.8. The generated flow graph is shown in figure 2.18. This flow graph starts with a `<bind>`, followed by a call to `*` with the arguments `a` and `2`. The result of the call is checked to be an `<integer>` by the `<multiple-value-check-type>`. A multiple value check is emitted because the generic `*` may return any number of values. The type check's result is returned.

## 2.8 Type Inference

The main task of type inference is to assign types to data flow nodes. This information is then used by the optimizer to allow optimizations like removal of type checks and upgrading of generic function calls. The algorithm described in this section is the original implementation, which was replaced by the algorithm described in section 4.3. The already existing type inference algorithm implemented is described in [Age96]. Type inference is done for each flow graph separately. This algorithm visits each control flow node and estimates a type for each data flow node. It contains rules for every computation class, and always conjoins the old type estimate with the new one, recording all rules contributing to it. If a type estimate changes, e.g. because an optimization was applied, all dependent type estimates are updated.

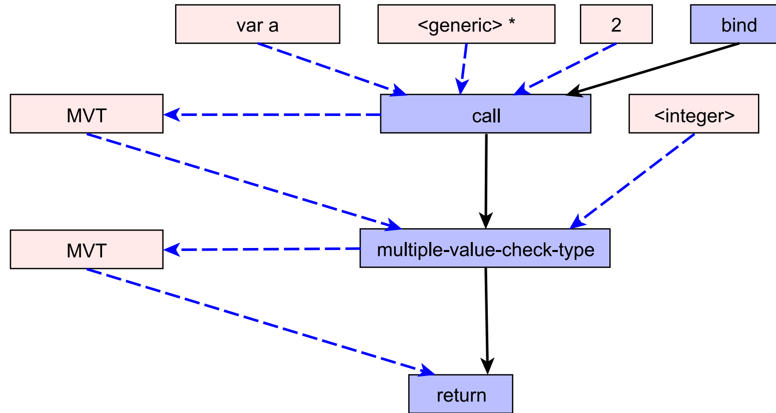


Figure 2.18: Flow graph of method `double`

To infer more narrow return types of methods, under presence of subtyping, an implementation of the cartesian product algorithm [Age95] is used. This infers (monomorphic) methods for different input types and saves the inferred output types for later reuse.

### 2.8.1 Drawbacks

Since polymorphic types were not representable by the type system in the past, some standard Dylan functions had hard-coded polymorphic types: The return type of `make` is its first argument, the return type of `element` is the element-type of a `limited` collection. These special treatments are unsafe, because the compiler does not detect if a developer specified (and actually returned) some other type for a `limited` collection or `make`.

From a maintainability and extensibility point of view another drawback of the implemented typist library was discovered. It contains a completely separate class hierarchy for `<type-estimate>`, analogous to the `<type>` hierarchy. Also, subtype and disjoint relations and operations like union, intersection, difference and normalization on `<type-estimate>` classes were implemented.

The former type inference algorithm did not take branches of the flow graph into account. Also, the type of variables with assignments was the union of the types of all assigned values. Both were reasons for constrained optimization possibilities and conservative type estimations.

## 2.9 Optimization

In production mode the compiler uses several optimizations to produce a more efficient (in both time and space dimensions) binary than in development mode. The implemented optimizations are dead code removal, constant folding, common subexpression elimination, inlining, dispatch upgrading and tail call analysis. Most of them are standard optimizations, described in detail in [ASU86]. Dispatch upgrading, which upgrades generic function calls to method calls, is

```
define open class <a> (<object>)
end;

define open class <b> (<object>)
end;
```

Listing 2.9: Classes <a> and <b>

```
define generic fun (a :: <object>);

define method fun (a :: <a>)
  ...
end;

define method fun (b :: <b>)
  ...
end
```

Listing 2.10: Function `fun`

non-standard, because most programming languages do not support multimethods. Therefore it is described in detail below.

### 2.9.1 Dispatch Upgrading

Generic function dispatch at run time is expensive, because all methods of the generic function have to be taken into account, applicability of argument types have to be checked and specificity has to be computed, and finally the most specific method has to be selected and called.

In order to do a generic function dispatch at compile time, all applicable methods of a generic function must be known. This is the case if the generic function is sealed, meaning it may not be extended elsewhere or at run time. The opposite of sealed is called open.

**Sealing** The motivation for sealing is limitation of dynamism to reason more static properties. The dynamism is caused by subtyping and extensibility described next.

If a type $a$ is a subtype of another type $b$, all instances of type $a$ are also instances of type $b$. Thus, whenever an instance of type $b$ is expected, an instance of type $a$ can be used (subsumption rule). In addition to the subtype relation, the disjoint relation is defined: Two types are disjoint if there cannot exist an object that is an instance of both types. Both relations are used for testing applicability of methods of a generic function. Due to dynamism and extensibility, the class hierarchy might only be partially known during compilation of a library, thus definition of both relations are useful. Even if $a$ is not a subtype of $c$ and $c$ is not a subtype of $a$, they might not be disjoint. Both relations are described in more detail in section 4.2.

The classes `<a>` and `<b>` are defined in library `foo`, as shown in listing 2.9.

```
define class <c> (<a>, <b>)
end;
```

Listing 2.11: Class <c>

```
define open class <a> (<object>)
end;

define class <b> (<object>)
end;
```

Listing 2.12: Classes <a> and <b>

Neither <a> nor <b> is a subtype of the other. But it is unknown whether they are disjoint, because there may exist a class <c>, shown in listing 2.11, in library bar, which is a subclass of both <a> and <b>.

The generic function fun, defined in listing 2.10, has methods defined on <a> and <b>. This generic function may not be extended elsewhere, because it is not declared open. Both methods are applicable for objects of class <c>. Calls to fun with argument type <a> or <b> may not be upgraded to a method call to the specific method, because an instance of <c> might be passed instead. Looking at the class precedence list (linearization of superclass graph, algorithm described in [Sha96, Chapter 5]) for class <c>, which is [<c>, <a>, <b>], fun (a :: <a>) is more specific than fun (b :: <b>) (described in detail in the next section) and must be called.

In listing 2.12, <b> is sealed. There may not exist any subclass of <b>, thus no class <c>. Class <a> and <b> are guaranteed to be disjoint. Calls to fun from listing 2.10 with argument types <a> or <b> may be upgraded to method calls. This applies analogue if class <a> is sealed.

If the generic function fun would be open, as shown in listing 2.13, upgrades from generic function calls to method calls are not possible, because fun may be extended in any library for subtypes of <a> or <b>. An example is shown in listing 2.14, where fun is extended for the singleton $empty-a$ (using == in an argument list specifies the singleton type containing the given object).

The extension of a generic function for subtypes of a set of argument types

```
define open generic fun (a :: <object>);

define method fun (a :: <a>)
  ...
end;

define method fun (b :: <b>)
  ...
end
```

Listing 2.13: Function fun

```
define constant $empty-a$ = make(<a>);

define method fun (a == $empty-a$)
  ...
end;
```

Listing 2.14: Function `fun`

```
define open generic fun (a :: <object>);

define method fun (a :: <a>)
  ...
end;

define sealed domain fun (<a>);

define method fun (b :: <b>)
  ...
end;

define sealed domain fun (<b>);
```

Listing 2.15: Function `fun`

can be forbidden, as shown in listing 2.15 by `define sealed domain` statements. In this case, specializing of subtypes of `<a>` and `<b>`, as with `$empty-a$` in listing 2.14, is not allowed. Calls with argument type `<a>` or `<b>` can be upgraded to method calls, if class `<a>` or `<b>` or both are sealed.

**Method specificity** Method specificity is defined in [Sha96, Chapter 6] and cited here for comprehension:

> To order two methods A and B with respect to a particular set of arguments, compare each of A's argument types with B's argument types in the corresponding position using the argument that was supplied for that position. The comparison works in the following way.
>
> - If the argument types are type equivalent (both are subtypes of each other), then A and B are unordered at the current argument position. That is, this argument position provides no information about the order of the two methods.
> - Otherwise, if the argument type of A is a subtype of the argument type of B, then A precedes B at the current argument position.
> - Otherwise, if both argument types are classes, then their order in the class precedence list of the argument's class is used to determine which is more specific. If A's argument type precedes

B's argument type in the class precedence list of the argument's class, then A precedes B at the current argument position.

- Otherwise, the methods are unordered in the current argument position.

The method A is more specific than the method B if and only if A precedes B in at least one argument position, and B does not precede A in any argument position. Similarly, B is more specific than A if and only if B precedes A in at least one argument position, and A does not precede B in any argument position. If neither of these cases apply then A and B are ambiguous methods.

### 2.9.2 Optimizer Flow

The optimization phase works on each flow graph separately. Initially, it converts all assigned variables into a `<cell>` representation.

Then, all control flow nodes are added to the optimization queue. Optimizations are applied in order until one is successful. For each element of that queue, dead code elimination and constant folding is attempted. Dispatch upgrading and inlining is tried on call computations afterwards. When new control flow nodes are inserted into the flow graph, they are added to the optimization queue, as well as consumers of the data flow nodes generated by those computations. The typist is notified when control flow nodes are added or removed, and updates its type estimate of the given data flow nodes.

After all control flow nodes have been optimized, common subexpressions are eliminated, tail calls are analyzed, and unused closures are removed.

### 2.9.3 Example

The optimized flow graph of method `double` (listing 2.8) is shown in figure 2.19. Since `2` and `a` are known to be of type `<integer>` (due to either being a literal or its type annotation in the signature), and `*` with two `<integer>` arguments is sealed, the generic function call to `*` can be upgraded to a method call to `*(<integer>, <integer>)`.

Next the method call is inlined, resulting in the flow graph shown in figure 2.20. At run time, Dylan objects are type tagged with two bits, where an integer have the tag bits set to 01. Integer literals prefixed by `%` in the figure are raw machine words, which are not tagged (the raw bits). To do a multiplication, first the argument `a` is cast to a machine word. Afterwards the tag bits are removed (done by `machine-word-logxor` with 1), it is multiplied by 2 (`machine-word-multiply-signal-overflow`) and the tag bits are added afterwards (`machine-word-logior` with 1). Finally the resulting machine word is cast back to an integer. The casts preserve type safety of the primitives, but do not occur in the generated binary code.

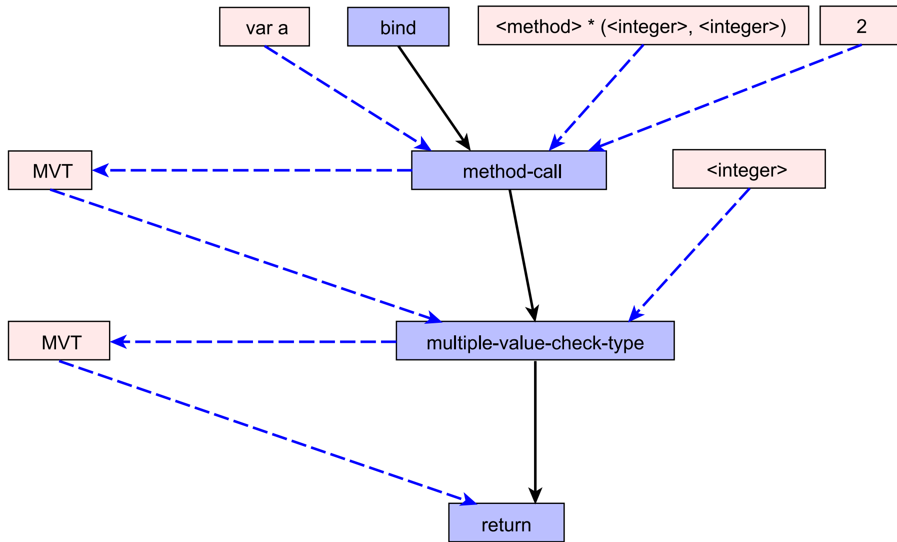


Figure 2.19: Flow graph of optimized method `double`

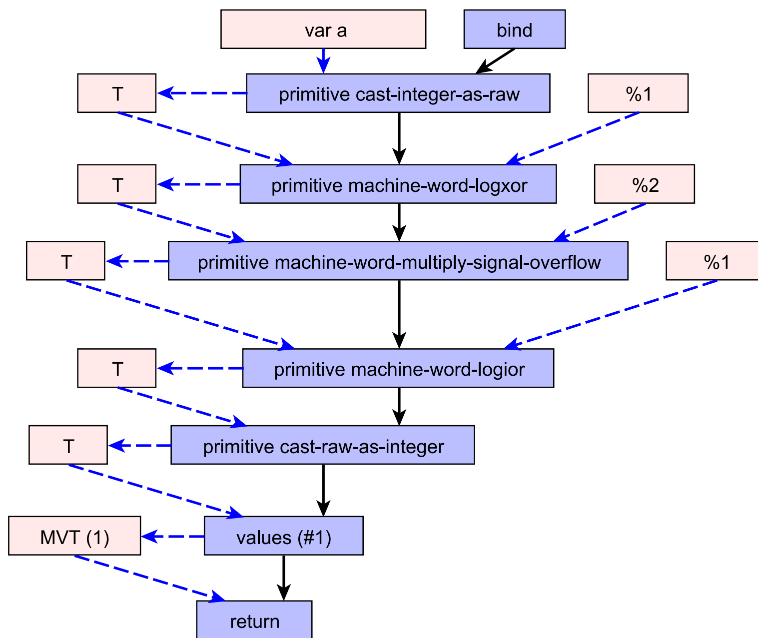


Figure 2.20: Flow graph of fully optimized method `double`

# Chapter 3

# Gradual Typing

Gradual typing is a concept developed by Jeremy Siek and Walid Taha [ST06] to support a smooth migration from dynamically to statically typed code. It extends the simply-typed $\lambda$ calculus ($\lambda_{\rightarrow}$) with a type representing the dynamic type, unknown at compile time, denoted by ?. The gradually-typed $\lambda$ calculus is written $\lambda^{?}_{\rightarrow}$.

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 (($\lambda$ (`x` : **number**) (succ x)) `#t`) should be rejected because the type of `#t` is not consistent with the type of argument `x`. That is, **boolean** is not consistent with **number**. On the other hand, the program (($\lambda$ (`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 run time by the application (succ `#t`).

Another example is mapping a function over a list, as shown in listing 3.1. The function `map` expects its first argument to be of type **number** $\rightarrow$ **number**, but the argument ($\lambda$ (x) (succ x)) has type ? $\rightarrow$ **number**. The type system should accept this program, so the intuition of type consistency of function types is that the known portions of the two types should be equal and the unknown portions to be ignored. The mathematics of partial functions contains an useful analogy: Two partial functions are consistent when every element that is in the domain of both functions is mapped to the same result. This analogy can be formalized by considering types as trees, as done in [Pie02, Chapter 21]. The ? represents places where the partial functions are undefined. Analogously to consistency of partial functions, the type consistency relation is reflexive and symmetric, but not transitive.

The type consistency relation is written $\sim$ and axiomatized with the definition in figure 3.1. In the following, $\tau$ will be used as a variable for any type, while $\gamma$ will be used for ground types, like **number** or **boolean**. Some examples

```
map : (number → number) × number list → number list
(map (λ (x) (succ x)) (list 1 2 3))
```

Listing 3.1: Mapping over a list

$$\gamma \sim \gamma$$

$$\tau \sim ?$$

$$? \sim \tau$$

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

Figure 3.1: Type consistency relation $\sim$

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** $\to$ ?.

The consistency relation for function types may seem strange on a first look, because it is not covariant and contravariant. But since consistency is symmetric, $\tau_1 \sim \tau_3$ is the same as $\tau_3 \sim \tau_1$.

The type system for $\lambda^?_\to$ is defined in [ST06] and shown in 3.2 ($g$ in $\vdash_g$ means gradual). The rules for variables, constants and functions are standard. Rule Var concludes that if $x$ is of type $\tau_1$ in context $\Gamma$, then $x$ is well typed at $\tau_1$ in $\Gamma$. Rule Cnst concludes that a constant $c$ is well typed in $\Gamma$ with its type. Rule Abs concludes that if $x$ is mapped to type $\tau_1$ and $e$ of type $\tau_2$ in context $\Gamma$, $\lambda x : \tau_1.e$ will be of type $\tau_1 \to \tau_2$. There are two rules for function application, rule App1 handles the case when the function type is unknown. The argument may have any type and the resulting type of the application is unknown. The second rule (App2) handles a function type and allows an argument whose type is consistent with the function's parameter type.

In [ST06] the relation of $\lambda^?_\to$ to the untyped $\lambda$ calculus is presented, together with a translation which converts any $\lambda$ term into an equivalent well-typed term of $\lambda^?_\to$. It is not possible to accept all terms of the untyped $\lambda$ 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 $\lambda$ 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 $\lambda$ calculus ($\lambda_\to$) is that both calculi are equivalent for terms of the $\lambda_\to$, proven in [ST06, Theorem 1]. A direct consequence of the equivalence is that the gradual type system catches the same static error as $\lambda_\to$.

In the following sections, approaches to inference of gradual typing are explored, as done in [SV08]. Afterwards the type system of $\lambda^?_\to$ with polymorphism is specified and an inference algorithm is presented (both also from [SV08]). In the final section of this chapter, the extension of subtyping to gradual typing is

$$\textsc{Var}\ \frac{\Gamma(x) = \tau_1}{\Gamma \vdash_g x : \tau_1}$$

$$\textsc{Cnst}\ \Gamma \vdash_g c : \mathit{typeof}(c)$$

$$\textsc{App1}\ \frac{\Gamma \vdash_g e_1 : ? \qquad \Gamma \vdash_g e_2 : \tau}{\Gamma \vdash_g e_1 e_2 : ?}$$

$$\textsc{App2}\ \frac{\Gamma \vdash_g e_1 : \tau_1 \to \tau_3 \qquad \Gamma \vdash_g e_2 : \tau_2 \qquad \tau_1 \sim \tau_2}{\Gamma \vdash_g e_1 e_2 : \tau_3}$$

$$\textsc{Abs}\ \frac{\Gamma(x \mapsto \tau_1) \vdash_g e : \tau_2}{\Gamma \vdash_g \lambda x : \tau_1 . e : \tau_1 \to \tau_2}$$

Figure 3.2: Type system for $\lambda^?_\to$

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

Listing 3.2: Example code fragment where `a` is of different types

given, which has been done in [ST07].

## 3.1 Design Considerations of Unification Based Inference

In [SV08] a unification based type inference algorithm for gradual typing is developed. Initially some straightforward approaches are considered, and examples of programs which should be well-typed but are rejected by the approach, or that should be ill-typed but are accepted.

The simply typed $\lambda$ calculus ($\lambda_\to$) is extended to the polymorphic $\lambda$ calculus ($\lambda^\alpha_\to$) by introduction of polymorphic type variables. These are handled by a substitution $S$, which is a mapping from type variables to types. Analogous $\lambda^?_\to$ is extended with type variables and substitution to $\lambda^{?\alpha}_\to$.

### 3.1.1 Dynamic Types as Type Variables

The first approach is to treat dynamic types as type variables. The resulting system is fully static, not gradual. The code shown in listing 3.2 behaves statically, because the inference algorithm would deduce from the function applications `f a` and `g a` that $\alpha$ = **number** and $\alpha$ = **boolean**. There is no solution for $\alpha$, thus the program would be rejected with a static error. However, the program can run without error in a dynamically typed language, and should type check in $\lambda^?_\to$.

```
let f (x : number) (g : ? → ?) = g x
```

Listing 3.3: Code fragment with dynamic type

```
let f (g : α) = g 1
f 1
```

Listing 3.4: Code fragment not well typed in static system

### 3.1.2 Ignore Dynamic Types During Unification

The second approach is to ignore dynamic types during unification. When considering the program in listing 3.3, the inference algorithm would deduce $? \to ? =$ **number** $\to \beta$, where $\beta$ is freshly allocated and represents the result value of the application `g x`. This decomposes to $? =$ **number** and $? = \beta$, where the latter is not processed further. So $\beta$ is an unsolved variable, giving the impression that `f` is polymorphic in $\beta$ and thus should behave uniformly for any choice of $\beta$. Suppose `g` to be the identity function (which type is $\alpha \to \alpha$, then `f` raises an error if $\beta =$ **boolean**, but not if $\beta =$ **number**.

### 3.1.3 Well-typed After Substitution

The third approach is well typed after substitution: A program is well typed in $\lambda^{\alpha}_{\to}$ if there is a substitution that makes the program well typed in $\lambda_{\to}$. Applying this to gradual typing, a program is well typed in $\lambda^{?\alpha}_{\to}$ if there exists a substitution which makes it well typed in $\lambda^{?}_{\to}$. This approach is too gentle, the code fragment in listing 3.4 requires $\alpha$ to be of type **number** and to be a function type at the same time. This code fragment is well typed when using the substitution $\alpha \mapsto ?$.

The problem with this third approach is that it can assign any type variable to ? and thereby all programs are well typed. To solve this problem, the type system must not add arbitrary ?, while it must allow propagation of ? that are annotated.

## 3.2 Type System for $\lambda^{?\alpha}_{\to}$

A type with more known types (and fewer unknown types) is more informative than a type with fewer known types (and more unknown types). It is required that the solution for a type variable is as informative as any type that constrains the type variable. This prevents a solution for a variable from introducing dynamic types that do not already appear in program annotations. Formally, the relation *less or equally informative*, written $\sqsubseteq$, is introduced in figure 3.3. The relation $\sqsubseteq$ is the partial order underlying the $\sim$ relation. In [SV08] it is shown that both relations can be defined in terms of the other.

Revisiting the example of listing 3.4, the fact **number** $\to \beta_0 \sqsubseteq \alpha$ is introduced by the application of `g` to `1` ($\beta_0$ is a fresh variable representing the result of the application). The application of `f` to `1` introduces the fact **number** $\to \beta_1 \sqsubseteq \alpha \to \beta_0$ which implies **number** $\sqsubseteq \alpha$ (due to $\to$ rule in fig 3.3). There is

$$? \sqsubseteq \tau$$

$$\gamma \sqsubseteq \gamma$$

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

Figure 3.3: $\sqsubseteq$ relation

$$\text{GVAR}\ \frac{\Gamma(x) = \tau_1}{S; \Gamma \vdash_g x : \tau_1}$$

$$\text{GCNST}\ S; \Gamma \vdash_g c : typeof(c)$$

$$\text{GAPP}\ \frac{S; \Gamma \vdash_g e_1 : \tau_1 \qquad S; \Gamma \vdash_g e_2 : \tau_2 \qquad S \vDash \tau_1 \simeq \tau_2 \rightarrow \beta \qquad (\beta fresh)}{S; \Gamma \vdash_g e_1 e_2 : \beta}$$

$$\text{GABS}\ \frac{S; \Gamma(x \mapsto \tau_1) \vdash_g e : \tau_2}{S; \Gamma \vdash_g \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2}$$

Figure 3.4: Type system for $\lambda_{\rightarrow}^{?\alpha}$

no solution for $\alpha$ that satisfies both **number** $\rightarrow \beta_0 \sqsubseteq \alpha$ and **number** $\sqsubseteq \alpha$, so the program is rejected.

The type system for $\lambda_{\rightarrow}^{?\alpha}$, which extends $\lambda_{\rightarrow}^{?}$ with type variables is shown in figure 3.4.

The rules for application changed: in $\lambda_{\rightarrow}^{?}$ there had been two rules, in $\lambda_{\rightarrow}^{?\alpha}$ only the rule GAPP is present. This is due to the fact that introducing extra ? is not allowed (section 3.1.3), and using a type variable in the substitution instead of a metavariable places this more strict requirement on the variable, a more detailed explanation is given in [SV08, 4.2 The Definition of the Type System]. The notation $S \vDash \tau_1 \simeq \tau_2$ is the consistent-equal relation under the substitution $S$.

## 3.3 An Inference Algorithm for $\lambda_{\rightarrow}^{?\alpha}$

A classic substitution based algorithm would not work, because it uses the first possible solution. Thus, with the constraint $\alpha \rightarrow \alpha \simeq\ ? \rightarrow$ **number**, ? would be substituted for $\alpha$, but later **number** $\not\sqsubseteq$ ?.

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

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

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

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

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

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

Figure 3.5: Constraint generation for $\lambda^{?\alpha}_{\to}$

This idea is integrated into Huet's almost linear algorithm [Hue76]. 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 composed nodes (like $\to$) to their children.

### 3.3.1 Constraint Generation

The constraint generation judgement has the form $\Gamma \vdash_g e : \tau | C$, where $C$ is the set of constraints. In figure 3.5 the constraint generation rules are given. The rules are tightly connected to the type system (figure 3.4), equivalence is shown in [SV08, Lemma 3]. The only rule which actually generates a constraint is CApp, where the side condition on the GApp becomes a generated constraint on the CApp rule.

For example, for the program ($\lambda$ f : **number** $\to$ **number** . $\lambda$ g : $\alpha$ . f g) the following constraint set is generated { **number** $\to$ **number** $\simeq \alpha \to \beta$ }, shown in figure 3.6.

For the program ($\lambda$ h : (**number** $\to$ ?) $\to$ (? $\to$ ?) $\to$ **number**. $\lambda$ y : $\alpha$ . h y y) the constraint set {(**number** $\to$ ?) $\to$ (? $\to$ ?) $\to$ **number** $\simeq \alpha \to \beta_1$, $\beta_1 \simeq \alpha \to \beta_2$ } is generated.

### 3.3.2 Constraint Solver

Huet's algorithm uses a graph representation for types. Every type is a node, a constraint is an edge between two nodes. For example in figure 3.6 the constraint **number** $\to$ **number** $\simeq \alpha \to \beta$ is shown. Constraint edges are dashed.

The definition of the algorithm `solve` is shown in listing 3.7. Its input is a set

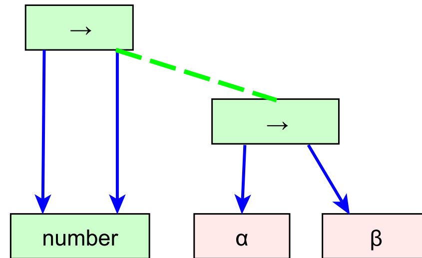


Figure 3.6: Constraint **number** → **number** $\simeq \alpha \rightarrow \beta$

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

Listing 3.5: Relation order

```
1  union(u,v,order_matters?) =
2    if order_matters? then
3      if u.rank = v.rank then
4        u.rank := u.rank + 1
5      v.representative := u
6    elseif u.rank > v.rank then
7      v.representative := u
8    else
9      u.representative := v
10     if u.rank = v.rank then
11       v.rank := v.rank + 1
```

Listing 3.6: Huet union

```
1  Input: Constraints C
2  C := copy_dyn(C)
3  for each node u do
4    u.contains_vars := true
5  while not C.empty
6    (x ≃ y) := C.get
7    u := find(x); v := find(y)
8    if u ≠ v then
9      (u,v,f) := order(u,v)
10     union(u,v,f)
11     case type(u) ≃ type(v) of
12       u1 → u2 ≃ v1 → v2 ⇒ C.add(u1 ≃ v1); C.add(u2 ≃ v2)
13       u1 → u2 ≃ ? ⇒
14         if u.contains_vars then
15           u.contains_vars := false
16           C.add(vertex(type=?, contains_vars=false) ≃ u1)
17           C.add(vertex(type=?, contains_vars=false) ≃ u2)
18       τ ≃ var | τ ≃ ? | γ ≃ γ ⇒ pass
19       _ ⇒ error: inconsistent types
20 G = quotient graph by equivalence class
21 if G acyclic then
22   return { u ↦ type(find(u)) | u a node in the original graph }
23 else error
```

Listing 3.7: Huet solve algorithm

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 11-19), where possibly more constraints are added. If the constraint set is empty, the quotient graph by equivalence class is constructed (line 20). 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 22). Otherwise an error is returned (line 23).

During merging a decision has to be made which node overrides which other node. This is done in the relation `order`, shown in listing 3.5. A type variable $\alpha$ 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`.

The actual merging is done in `union` (listing 3.6), which takes two nodes and a flag as input (line 1). If the flag is true, the first node 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). Again, the rank is only increased if the rank of both nodes is equal (lines 10, 11).

The case analysis of `solve` (lines 11-19) consists of several cases:

- The first case (line 12) 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 13-17) 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 16 and 17).
- The third case (line 18) 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. These constraints pass.
- If none of those cases matches, an error is reported (line 19).

Consider the program $(\lambda\, \mathrm{f} : ?.\ \lambda\, \mathrm{x} : \alpha\ .\ \mathrm{f\,x})$, where the constraint $? \simeq \alpha \to \beta$ is generated, $\beta$ is the result of the application. Thus, `f` is "cast" to $? \to ?$, thus we have the constraint $? \to ? \simeq \alpha \to \beta$, which is then solved to the constraints $? \simeq \alpha$ and $? \simeq \beta$. If the type of $\alpha$ is $\alpha \to \alpha$, an infinite loop would occur in the solver (by replacing $? \simeq \alpha$ with $? \to ? \simeq \alpha \to \alpha$, which is solved to $? \to \alpha$ 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 3.2 again, which generates (by applying rule CApp twice) the constraint set $\{\ \mathbf{number} \to ? \simeq ? \to \beta_0, \mathbf{boolean} \to ? \simeq ? \to \beta_1\}$ in the last line, where ? is the identical node.

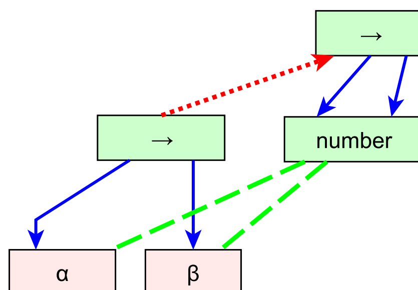


Figure 3.7: Type graph after first constraint solved, $C$ = { **number** $\simeq \alpha$, **number** $\simeq \beta$ }

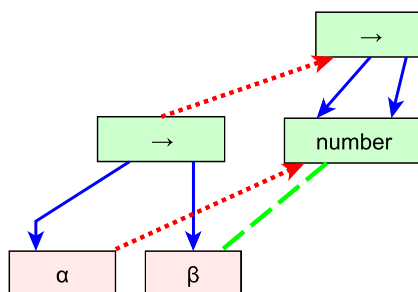


Figure 3.8: Type graph with last constraint, $C$ = { **number** $\simeq \beta$ }

It is used as return value of `f` and `g` and as type of `a`. The type variables $\beta_0$ and $\beta_1$ are freshly allocated by the CApp rule, and represent the return value of `f` and `g`. Solving these constraints results in the contradictory constraints **number** $\simeq$ ? and **boolean** $\simeq$ ?. Thus, an error is reported. To prevent from this error, the function `copy_dyn` replaces all ? with separate nodes, which can then be unified with different types.

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 23). Otherwise the mapping of type variable to type is returned (line 22). An example where the quotient graph contains cycles is $\lambda x.xx$. The type of $x$ is constrained with a function type (due to the application) which argument is the type of $x$.

An example (already presented in the constraint generation phase: $\lambda$ f : **number** $\rightarrow$ **number** . $\lambda$ g : $\alpha$ . f g) of the solve algorithm with the input constraint set { **number** $\rightarrow$ **number** $\simeq \alpha \rightarrow \beta$ } is shown in figure 3.6. During the first iteration, `order` returns (**number** $\rightarrow$ **number**, $\alpha \rightarrow \beta$, false) (line 5 matched), thus the unifier sets the representative of the $\alpha \rightarrow \beta$ to **number** $\rightarrow$ **number** (line 8 of `union`). The first case of the solver matches (line 12), thus two new constraints are pushed, { **number** $\simeq \alpha$, **number** $\simeq \beta$ }, as shown in

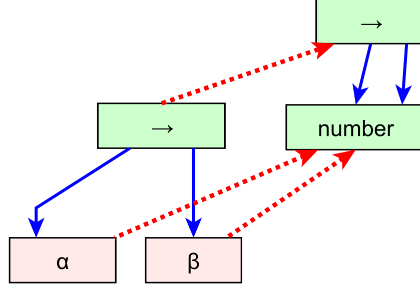


Figure 3.9: Final type graph, { $\alpha \mapsto$ **number**, $\beta \mapsto$ **number** }

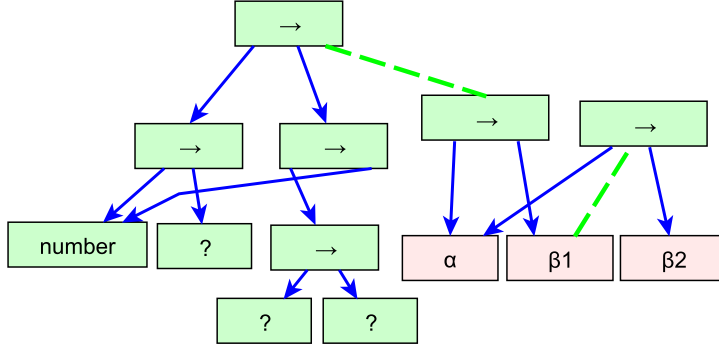


Figure 3.10: Initial type graph, $C$ = { (**number** → ?) → (? → ?) → **number** $\simeq \alpha \rightarrow \beta_1$, $\beta_1 \simeq \alpha \rightarrow \beta_2$ }

3.7 (representative edges are dotted). Now **number** $\simeq \alpha$ is solved, where `order` returns (**number**, $\alpha$, true) (line 4 matched), thus `union` sets the representative of $\alpha$ to **number**. The third rule of the case in the solve algorithm matches (line 18), thus nothing more happens. This is shown in figure 3.8. The constraint **number** $\simeq \beta$ is solved in the same way, the representative of $\beta$ is now also **number**. The resulting graph is shown in 3.9. The quotient graph consists of the representative nodes (**number** and **number** → **number**) with two edges from **number** → **number** to **number**. This quotient graph does not contain any cycles. Thus, the mapping { $\alpha \mapsto$ **number**, $\beta \mapsto$ **number** } is returned.

Another example (also presented in the constraint generation phase: $\lambda$ h : (**number** → ?) → (? → ?) → **number**. $\lambda$ y : $\alpha$ . h y y) with the input constraint set {(**number** → ?) → (? → ?) → **number** $\simeq \alpha \rightarrow \beta_1$, $\beta_1 \simeq \alpha \rightarrow \beta_2$ } is shown in figure 3.10. The first iteration removes the → constraint and adds two new constraints, one where the arguments of each → are constrained, the other constrains the values. The resulting constraint set is {**number** → ? $\simeq \alpha$, (? → ?) → **number** $\simeq \beta_1$, $\beta_1 \simeq \alpha \rightarrow \beta_2$ }, shown in figure 3.11. The first constraint sets the representative of $\alpha$ to **number** → ?, the second sets the representative of $\beta_1$ to (? → ?) → **number**, shown in figure 3.12. The final constraint is, after `find` replaced the nodes with their representatives:

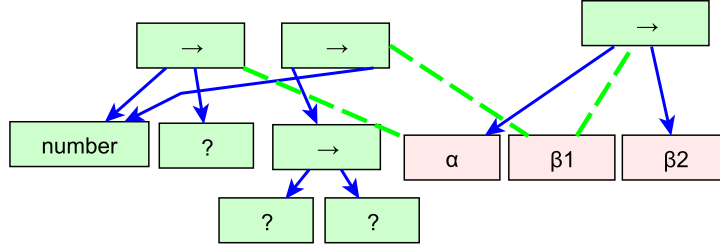


Figure 3.11: Type graph with constraint set, $C = \{\ \textbf{number} \rightarrow ?\ \simeq\ \alpha$, $(? \rightarrow ?) \rightarrow \textbf{number} \simeq \beta_1, \beta_1 \simeq \alpha \rightarrow \beta_2\ \}$

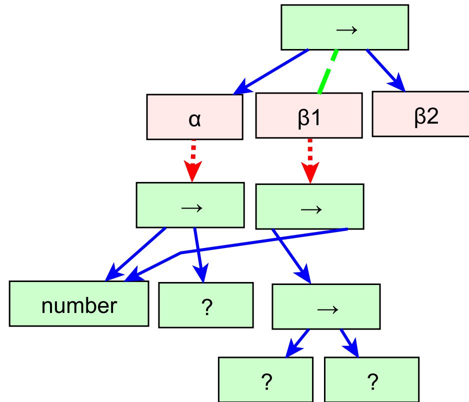


Figure 3.12: Type graph with remaining constraint, $C = \{\ \beta_1 \simeq \alpha \rightarrow \beta_2\ \}$

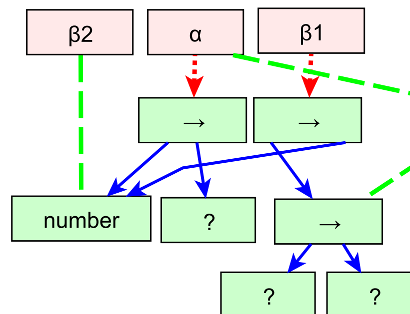


Figure 3.13: Type graph with constraint set $C = \{\ ? \rightarrow ? \simeq \alpha, \textbf{number} \simeq \beta_2\ \}$

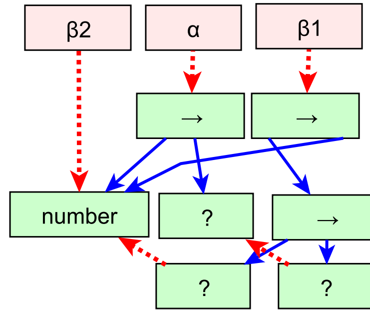


Figure 3.14: Final type graph, { $\alpha \mapsto$ **number** $\rightarrow$ ?, $\beta_1 \mapsto$ **number** $\rightarrow$ ?, $\beta_2 \mapsto$ **number** }

(? $\rightarrow$ ?) $\rightarrow$ **number** $\simeq \alpha \rightarrow \beta_2$. It is again solved by constraining arguments and values of each $\rightarrow$ node, resulting in the constraint set { ? $\rightarrow$ ? $\simeq \alpha$, **number** $\simeq \beta_2$ }, shown in figure 3.13. Since the representative of $\alpha$ is **number** $\rightarrow$ ?, again two constraints are pushed, during solution of those ? both times gets a new representative. The final type graph is shown in figure 3.14. Since the quotient graph does not contain cycles, the mapping { $\alpha \mapsto$ **number** $\rightarrow$ ?, $\beta_1 \mapsto$ **number** $\rightarrow$ ?, $\beta_2 \mapsto$ **number** } is returned.

## 3.4 Subtyping

The integration of gradual typing and subtyping was developed in [ST07], which applied gradual typing to the object calculus ($Ob^{<:}$) of Abadi and Cardelli [AC], resulting in $Ob^{?}_{<:}$. There are two different kinds of subtyping, nominal and structural. While the former applies to type systems where a developer explicitly writes down superclasses of a given class (as done in Java, Dylan, etc.), and thus defines the subtype relation of classes, the latter looks at the structure of types (as done in JavaScript, Ocaml, etc.), and two different classes with different names but same structure are considered to be equal. Both have advantages and disadvantages, described in detail in [Pie02, Chapter 19]. Gradual typing with subtyping as described in [ST07] and summarized in this section was done in a structural type system.

The main idea is that uses of subtyping in the type checker are replaced with a relation which integrates subtyping with type consistency. If ? is treated as top of the subtype hierarchy, $? <: S$ would hold for any $S$ (allowing implicit down-cast from top to any type). Also, $R <:?$ would be true (normal up-cast rule). Using transitivity $R <: S$ would hold for all types $R, S$. Therefore the type system would not reject any program.

Thus ? is treated neutral to subtyping, a single subtype rule matches ?, reflexivity (S-REFL) $? <: ?$. A type system for $Ob^{?}_{<:}$ is defined in [ST07]. More attention has to be taken for a type checking algorithm and operational semantics, both are also presented in the paper.

$Ob^{?}_{<:}$ is not explained in detail here, common properties (soundness of cast insertion, type safety, progress, preservation, etc.) are proven in [ST07].

# Chapter 4

# Type System for Dylan

In this chapter I will outline what is needed to implement a type system for Dylan. After deducing type system features necessary for Dylan's type constructors, I will define important type relations: subtyping, which will be extended for all features of the type system, and disjointness, used mostly for optimization of generic function calls. Afterwards I will extend the type inference algorithm described in section 3.3 with the necessary features to be applicable to Dylan. Finally I will discuss the interaction of generic functions and parametric polymorphism.

Dylan is object oriented and has a nominal type system: class inheritance is declared explicitly. While most object oriented programming languages employ the Object calculus ($Ob_{<:}$, [AC]) to model their theoretical foundations, Dylan's type system is different. Methods are defined outside of classes and class inheritance's effects are inheritance of member variables in the data structure and construction of a subtype relation between subclass and its superclasses (Dylan supports multiple inheritance).

Thus the type system implemented here is based on $\lambda_{\rightarrow}^{?\alpha}$, described in previous section 3. Most of the extensions applied to $\lambda_{\rightarrow}^{?\alpha}$ are described in standard literature ([Pie02]). As of now there are formal proofs neither for properties (like subject reduction, progress, preservation) for this extended type system nor for correctness and completeness of the implementation.

In this section Dylan's original type system, until now only described informally in [Sha96, Chapter 5], is described and parts are formalized. A Dylan program is well-typed, if it neither contains run time type checks nor generic function calls. Both are emitted during conversion phase where needed (lexical variables with type annotations, return value type annotations, calls). In the type inference phase the compiler can remove redundant type checks and upgrade generic function calls to specific method calls (section 2.9.1).

## 4.1 Type Constructors

The following types are constructable in Dylan (from [Sha96, Chapter 5]). The types form a hierarchic structure, with a dedicated element at the top and one at the bottom, called lattice.

- $\top$ - the top type, top of the type hierarchy, the same as `<object>`

- $\perp$ - the bottom type, bottom of the type hierarchy
- class - every class definition defines a corresponding type - define class <foo> (<object>) defines a class <foo> with superclass <object>
- union - union of any number of types - type-union(<integer>, <string>) specifies the type which may be either a <string> or an <integer>
- singleton - a type containing a single object - singleton(#f) is the type of the single object false
- limited types - a restricted type, there are three specified types which may be limited:
  - limited collection type - the element-type and size of a collection may be restricted - limited(<vector>, of: <integer>, size: 3) specifies the type of a vector which size is exactly three elements and contains only <integer> instances
  - limited integer - an integer with a more precise range - limited(<integer>, min: 0, max: 16) is the type of an <integer> between 0 and 16; limited(<integer>, min: 0) is the type of natural numbers
  - subclass - a class and all its subclasses - subclass(<number>) is the set of classes which are numbers: <number>, <integer>, <float>, etc.

In order to specify function types more precisely and to make use of parametric polymorphism, a feature already present in $\lambda^{?\alpha}_{\rightarrow}$, I added the following types to Dylan:

- polymorphic type variable - bounded quantification, a type variable with an optional upper bound - define method id (forall: A) (x :: A) => (x :: A) specifies the method id with a polymorphic type
- limited function type - argument and return values of a function may be specialized - define method foo (x :: <integer> => <string>) specifies an argument x of the function type with exactly one argument of type <integer> and one result value of type <string>.

## 4.2 Relations on Types

As mentioned above, certain analyses require knowledge about types that may partly be only available at run time, e.g. due to run time extension of classes that come with Dylan's dynamism. Still, the compiler needs and has enough information to reason about most types at compile time. After all, it should report type inconsistencies and upgrade generic functions (see section 2.9.1). Therefore I now describe in detail both relations the compiler needs for analysis, subtype and disjointness.

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \text{ T-SUB}$$

Figure 4.1: Subsumption rule

### 4.2.1 Subtype

Subtyping [Pie02, Chapter 18] is often considered an essential feature in object-oriented programming languages. It is useful for separating an object's type from its internal representation. For the compiler an object's type, also called its interface, is just a set of operations, each consisting of a name and a signature. With subtyping in mind, if an object implements an interface $S$, it also implements the interface $T$ with fewer operations. Using an object with interface $S$ at places where an object of type $T$ is expected should always be safe. This is formalized in the simply typed $\lambda$ calculus with subtyping ($\lambda_{<:}$) by the subsumption rule T-SUB, shown in figure 4.1. Subsumption allows to use $t$ of type $S$, if $S$ is a subtype of $T$ (denoted by $<:$), where a value of type $T$ is expected.

The subtype relation, as defined in [Pie02, Chapter 15] is shown in figure 4.2. The subtype relation is reflexive (rule S-REFL) and transitive (rule S-TRANS). It has a special element $\top$, which is the supertype of all types (rule S-TOP). Function types must be contravariant in the arguments and covariant in the values, as seen in the rule S-ARROW. If the domain $S_1$ is bigger than the domain $T_1$ ($T_1 <: S_1$) and the codomain $S_2$ is smaller than the codomain $T_2$ ($S_2 <: T_2$), the function $S_1 \rightarrow S_2$ is a subtype of $T_1 \rightarrow T_2$.

$\top$ The type $\top$ is equal to the type `<object>` in Dylan.

$\rightarrow$ The limited function type corresponds to an $\rightarrow$ type, which is covered by the rule S-ARROW.

$\perp$ A straightforward extension to the basic subtyping relation is the bottom element $\perp$ ([Pie02, Chapter 15]). The bottom element is the subtype of all types, as shown in figure 4.3. It is useful to express the type of operations not returning any value.

**Class** Every class definition $C$ explicitly specifies its direct superclasses $D_1 \ldots D_n$, shown in rule S-CLASSDEF in figure 4.4. Every superclass of every $D_i$ is also a superclass of $C$, as shown in rule S-TRANS in figure 4.2. According to rule S-REFL, every class $C$ is also a subtype of itself.

**Union** The Dylan type constructor `type-union` generates an union of all input types. Subtyping of type unions in Dylan are described informally in [Sha96, Chapter 5]. Formal subtype rules for union types were taken from [BP99] and are shown in figure 4.5. If both $R$ and $S$ are subtypes of $T$, the union $R \cup S$ is also a subtype of $T$ (rule S-UNION-L). Every member of a union $S_i$ is a subtype of the union $S_1 \cup S_2$ (rule S-UNION-UB). Trivially,

$$S <: S \quad \text{S-Refl}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{S-Trans}$$

$$S <: \top \quad \text{S-Top}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad \text{S-Arrow}$$

Figure 4.2: Subtyping relation

$$\bot <: T \quad \text{S-Bot}$$

Figure 4.3: Subtyping rule for $\bot$

$$\frac{define\ class\ C(D_1 \ldots D_n)}{C <: D_1 \ldots C <: D_n} \quad \text{S-classdef}$$

Figure 4.4: Subtyping rule for classes

$$\frac{R <: T \qquad S <: T}{R \cup S <: T} \text{ S-UNION-L}$$

$$S_i <: S_1 \cup S_2 \text{ S-UNION-UB}$$

Figure 4.5: Subtyping rules for union types

$$\frac{\Gamma \vdash m : B}{\Gamma \vdash S(m) <: B} \text{ S-SINGLETON-BASE}$$

$$\frac{\Gamma \vdash m_1 = m_2 : B}{\Gamma \vdash S(m_1) <: S(m_2)} \text{ S-SINGLETON}$$

Figure 4.6: Subtyping rules for singleton types

`type-union(a, b, c)` can be written as `type-union(a, type-union(b, c))`, thus extension of the rules to unions with more than two elements is straightforward.

**Singleton** To represent types with exactly one value (like the type of false) the `singleton` type constructor is provided. Singleton types have been investigated in [SH06], the depicted rules are shown in figure 4.6. In contrast to previous subtyping rules, the rules for singletons need an environment $\Gamma$, because a singleton is constructed by a real object which exists in the environment. The rule S-SINGLETON-BASE shows if there is an object $m$ of type $B$ (its base type), then the singleton type $S(m)$ is a subtype of type $B$. Rule S-SINGLETON shows the subtype relationship between two singleton types. If $m_1$ is identical to $m_2$, then the singleton type $S(m_1)$ will be a subtype of the singleton type $S(m_2)$.

Subtyping of singletons and limited types is defined differently in [Sha96]. A case not covered by the rules in figure 4.6 is `singleton(2)` <: `limited(<integer>, min: 0, max: 3)`. In [Sha96], a check whether the singleton object is an instance of the given type is done as subtype test. The correlation between subtype and instance is defined in [Sha96] for an object $m$ and a type $T$: If $m$ is an instance of $T$, the singleton type of $m$ is a subtype of $T$. This holds for both directions. Since Dylan contains run time type information, an instance check can be done easily.

**Limited integer** Limited integers, defined in [Sha96, Chapter 5], are always subtypes of `<integer>`. The subtype rules are given in figure 4.7, where

$$int_{min}^{max} <: <integer> \text{ S-LIMINT-BASE}$$

$$\frac{min_a \geq min_b \qquad max_a \leq max_b}{int_{min_a}^{max_a} <: int_{min_b}^{max_b}} \text{ S-LIMINT}$$

Figure 4.7: Subtyping rules for limited integer types

$int_{min}^{max}$ represents all integers between $min$ and $max$. Those rules are not in the standard literature, but intuition is straightforward that any integer with bounds is a subtype of the integer class (rule S-LIMINT-BASE). The rule S-LIMINT follows the intuition of set theory that where an object of type $int_0^{16}$ is expected, an object of type $int_{10}^{12}$ is also accepted, but an object of type $int_{12}^{17}$ is not accepted, because 17 is member of $int_{12}^{17}$, but not of $int_0^{16}$.

**Limited collection** Limited collection, from [Sha96, Chapter 8], allow narrower specifications of collection. Their element type $t$ as well as their size $s$ can be specified ($lcoll_{c,s}(t)$), where $c$ is the class implementing the collection ($c <: <collection>$). The subtype rules are shown in figure 4.8. The rule S-LCOLL-BASE shows that every limited collection is a subtype of the collection narrowed down. The rule S-LCOLL1 is based on the intuition that a collection $lcoll_{C_1,n}(T_1)$ of any type $C_1$ is a subtype of another collection $lcoll_{C_2,m}(T_2)$ if the collection type is a subtype ($C_1 <: C_2$) and the element types are equivalent ($T_1 <: T_2$, $T_2 <: T_1$). This can be directly derived from [Pie02, Chapter 15]. Additionally, if a size is declared, this must be identical in both limited collections, rule S-LCOLL2. This size equivalence is specified in [Sha96, Chapter 8] and is counterintuitive: a collection with a smaller size should be a subtype of a collection with a bigger size (as subtype relation of record types [Pie02, Chapter 11]).

Arrays are collections whose elements are arranged according to a Cartesian coordinate system. Single-dimensional arrays are called vectors. For multi-dimensional arrays dimensions can be passed as a sequence of integers: $d$ ($larr_{C,d}(T)$, where $C <: <array>$). The rule S-LARRAY1 handles multi-dimensional arrays with specified dimensions, both dimensions must be identical. If the subtype specified dimensions and the supertype did not, the size $m$ of the supertype must be equal to the product of the dimensions, as shown in rule S-LARRAY2. These rules do not occur in literature. The rule S-LARRAY2 is again counterintuitive: a vector of size 4 is a subtype of a two-dimensional array with dimensions `[2, 2]`.

**Polymorphic type variables** Universally quantified type variables without any bound are of $\top$ type, while if they contain a bound, subtyping is defined on this bound. The paper [DGLM95] discusses the problem of subtyping and parametric polymorphism, and defines subtyping only for

$$lcoll_{C,n}(\tau) <: C \text{ S-LCOLL-BASE}$$

$$\frac{C_1 <: C_2 \qquad T_1 <: T_2 \qquad T_2 <: T_1 \qquad m = \#f}{lcoll_{C_1,n}(T_1) <: lcoll_{C_2,m}(T_2)} \text{ S-LCOLL1}$$

$$\frac{C_1 <: C_2 \qquad T_1 <: T_2 \qquad T_2 <: T_1 \qquad m \equiv n}{lcoll_{C_1,n}(T_1) <: lcoll_{C_2,m}(T_2)} \text{ S-LCOLL2}$$

$$\frac{C_1 <: C_2 \qquad T_1 <: T_2 \qquad T_2 <: T_1 \qquad d_1 \equiv d_2}{larr_{C_1,d_1}(T_1) <: larr_{C_2,d_2}(T_2)} \text{ S-LARRAY1}$$

$$\frac{C_1 <: C_2 \qquad T_1 <: T_2 \qquad T_2 <: T_1 \qquad \prod d_1 \equiv m}{larr_{C_1,d_1}(T_1) <: larr_{C_2,m}(T_2)} \text{ S-LARRAY2}$$

Figure 4.8: Subtyping rules for limited collection types

$P_1[T] <: P_2[T]$, where $T$ is a polymorphic type variable. But there is no $P[S] <: P[T]$ rule if $S <: T$, because this most often is not the desired behavior (it would be legal to pass a collection of $S$ where a collection of $T$ is expected, and the function could insert objects of type $T$ in the collection of $S$). This is already covered by the subtyping rules of limited collections.

### 4.2.2 Disjointness

Two types are disjoint if neither can be a subtype of the other. Disjointness is symmetric. It is a conservative estimation, for there might be more types found being disjoint from each other at run time than figured out at compile time. As described in section 2.9.1, disjointness is used to filter the non-applicable methods of a generic function applied to actual arguments. In the following, $\tau$ is used as a metavariable for any type.

**($\top$, $\tau$)** No type $\tau$ is disjoint from $\top$.

**($\bot$, $\tau$)** Every type $\tau$ is disjoint from $\bot$, apart from $\bot$.

**(class A, class B)** Two classes are disjoint if they have no common subclass.

**(union, $\tau$)** A type-union $R \cup S$ is disjoint from any type $\tau$ if both $R$ is disjoint from $\tau$ and $S$ is disjoint from $\tau$.

**(singleton, $\tau$)** A singleton $S(m : B)$ is disjoint from type $\tau$ if $B$ is disjoint from $\tau$. Limited types complicate this relation, again an instance check is needed for those cases: The singleton type of the value 2 (which base type is `<integer>`) is not disjoint from the limited integer type $int_0^5$.

**($\alpha$, $\tau$)** A type variable $\alpha <: S$ is disjoint from $\tau$ if its upper bound $S$ is disjoint from type $\tau$.

**(limited integer, $\tau$)** A limited integer is disjoint from any type $\tau$ which is neither `<integer>` nor another limited integer.

**(limited integer, limited integer)** Two limited integers, $int_{min_a}^{max_a}$ and $int_{min_b}^{max_b}$ are disjoint if either $min_a > max_b$ or $min_b > max_a$.

**(limited collection, $\tau$)** Every limited collection is disjoint from any other type $\tau$, unless $\tau$ is the base collection class or another limited collection.

**(limited collection, limited collection)** A limited collection $lcoll_{C_1,n}(\tau_1)$ is disjoint from $lcoll_{C_2,m}(\tau_2)$ if $C_1$ is disjoint from $C_2$ or $\tau_1 \not<: \tau_2$ or $\tau_2 \not<: \tau_1$ or $n \neq m$. This also applies for limited arrays if $n$ and $m$ are computed by the product of the dimensions, if not provided.

**(limited function, $\tau$)** A limited function is disjoint from any other type $\tau$, unless $\tau$ is the class `<function>` or another limited function.

**(limited function, limited function)** A limited function $A \to B$ is disjoint from a limited function $C \to D$ unless $A \to B <: C \to D$ or $C \to D <: A \to B$ holds.

## 4.3 Extensions to Type Inference Algorithm

The type inference algorithm described in section 3.3 is extended to match the demands Dylan has. Both constraint generation and solve algorithm needed extensions. The former to cope with the different control flow graph structures described in section 2.6. The latter to cope with type system extensions: subtyping, tuple types, record types, described at the beginning of this chapter. Those are described in detail in the following.

### 4.3.1 Constraint Generation

The basic constraint rules (shown again in figure 4.9) described in section 3.3.1 are applicable to some control flow nodes. Other control flow nodes do not match these rules, thus more constraint rules are developed within this section.

**Binding** Binding computations (`<temporary-transfer>`) apply rule C-VAR, thus output and input are of same type.

**Type checks** Type checks assign the expected type ($\tau_2$) to the generated temporary. This is shown in rule C-TYPECHECK in figure 4.10. The rule is caused by the semantics of the type check, it will error if the value is not a subtype of the expected type ($\tau_1 \not<: \tau_2$). The same applies for multiple value type checks, where the expected type is a tuple type and the generated temporary is a multiple value temporary.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_g x : \tau | \{\}} \text{ C-VAR}$$

$$\Gamma \vdash_g c : typeof(c) | \{\} \text{ C-CNST}$$

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

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

Figure 4.9: Constraint generation for $\lambda^{?\alpha}_{\to}$

$$\frac{\Gamma \vdash_g x : \tau_1 | C \quad \tau_1 <: \tau_2}{\Gamma \vdash_g typecheck(x, \tau_2) : \tau_2 | C} \text{ C-TYPECHECK}$$

Figure 4.10: Constraint generation rules for type checks

$$\frac{\Gamma \vdash_g l_1 : \tau_1 \ldots \Gamma \vdash_g l_n : \tau_n | C}{\Gamma \vdash_g values(l_1 \ldots l_n) : (\tau_1 \times \ldots \times \tau_n) | C} \text{ C-VALUES}$$

$$\frac{\Gamma \vdash_g values(l_1 \ldots l_n) : (\tau_1 \times \ldots \times \tau_n) | C}{\Gamma \vdash_g extract_j(l_1 \ldots l_n) : \tau_j} \text{ C-EXTRACT-VALUE}$$

Figure 4.11: Constraint generation rules for multiple values

**Guarantee type** Because guarantee type may introduce unsafety, if the type the developer provided is used unchecked in the compiler, this computation is not handled in type inference.

**Variable reference** The type of the temporary generated by a variable reference computation is of the type of the referred variable.

**Multiple values** Multiple values need some special support, shown in figure 4.11. The rule C-VALUES supports the `<values>` computation by adding a constraint that the generated multiple value temporary is equal to the `<tuple>` of all types of the input values. The other rule, C-EXTRACT-VALUE, support the `<extract-value-computation>` computation and constrains the tuple at the given index with the generated temporary.

When the rest value of a variable arity temporary is extracted, the tuple type of the rest value is used, since the rest value are multiple values.

The adjustment of a multiple value temporary also adjusts the type in the same manner: If the adjustment shortens the length of a multiple value temporary, also the type of the generated temporary is the type of the input multiple value tempoary shortened by the length.

Both the backup and restore operations do not invoke any type rules, because they do not introduce any data flow nodes.

**Application** Application (`<call>`) computations apply rule C-APP (figure 4.9), a type graph $\rightarrow$ node consisting of a tuple type which consists of the argument types and a tuple type which consists of the value types is constrained with a $\rightarrow$ node representing the signature of the function.

The signature of a function consists of a fixed number of required arguments and values, and may contain a variable arity argument (`#rest`). Additionally a record type representing the keyword parameters may be present, which is currently ignored.

**Stack vector** Depending on the calling convention (congruent or non-congruent call) where the `<stack-vector>` is used, either a record type with keywords and values is generated, or a tuple type with uniform or non-uniform types (of fixed or variable arity). The binary representation of the generated temporary is always a vector, it is a contiguous memory region where all input values are located, one after another.

$$\frac{\Gamma \vdash_g e_1 : \tau_1 | C_1 \qquad \Gamma_c \vdash_g e_2 : \tau_2 | C_2 \qquad \Gamma_a \vdash_g e_3 : \tau_3 | C_3 \qquad \Gamma(\tau) = \Gamma_c(\tau_2) \cup \Gamma_a(\tau_3) \qquad \Gamma_c \supset \Gamma \qquad \Gamma_a \supset \Gamma}{\Gamma \vdash_g if\ e_1\ then\ e_2\ else\ e_3\ end : \tau | C_1} \text{C-IF}$$

$$\frac{\Gamma_1 \vdash_g e_1 : \tau_1 | C_1 \qquad \Gamma_2 \vdash_g e_2 : \tau_2 | C_2}{\Gamma \vdash_g merge(e_1, e_2) : \Gamma_1(\tau_1) \cup \Gamma_2(\tau_2) | \{\}} \text{C-MERGE}$$

Figure 4.12: Constraint generation rules for conditional

Thus, on the one side, the type must be a vector, subsequent users can for example copy it with `primitive-copy-vector`, on the other side, depending on the calling convention, it represents either a tuple or a record type.

**Abstraction** Abstraction (`<make-closure>`) adds the constraint that the generated temporary must be type-consistent with a $\rightarrow$ node containing the signature of the closure, which is formalized in rule C-Abs (figure 4.9).

**Conditional** The conditional `<if>` instantiates type environments for both branches, which inherit the type environment of the conditional as their parent type environment, as described later in section 5.7.2. The typing rule C-if is shown in figure 4.12.

**Merge** Merge computations add the type of the generated temporary to be the union of the types of both input data flow nodes (in their respective type environments). This is shown in rule C-merge in figure 4.12.

**Loop** The type inference algorithm of a loop construct searches for a fix point for all loop variables. This fix point is computed by assigning the outer type of each loop variable to output of the merge computation. Afterwards type inference is applied to the loop body and if the resulting inner loop types are equivalent to the outer types, these are used. Otherwise $\top$ is used.

This is safe as long as there is no generic function call inside of the loop body which depends on loop variables, because this might lead to different methods being called during each loop iteration.

A simple loop is shown in listing 4.1, where the outer type is `<integer>`, and can be safely used, because the resulting inner type is also `<integer>` and no generic function call depending on `i` is done in the loop body.

In the loop shown in listing 4.2, `i` cannot safely be inferred to be of type `<integer>`, which holds for the first two iterations. Because in the third iteration of the loop `i` will get assigned to the string "foo". The call to the generic `foo` must not be upgraded to a direct method call.

**Blocks** The body of a block is inferred as usual, but the resulting types are not used afterwards, because execution may have been interrupted by a

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

Listing 4.1: Safe loop

```
define method foo (x)
  "foo"
end;

define method foo
 (x :: limited(<integer>, max: 25))
  x + 10
end;

let i = 0;
while (i < 42)
  i := foo(i)
end
```

Listing 4.2: Unsafe loop

condition or the named exit. In unwind-protect blocks, which contain a sequence of cleanup computations, type inference is done as usual with the results being used afterwards, because the cleanup computations are guaranteed to be executed.

**Slots** If a call has been upgraded to a direct slot access, type safety of the arguments is already guaranteed. The generated temporary of a reading and writing slot access is of the same type as the accessed slot.

**Assignment** The `<set!>` computation is applicable if the types of value is a subtype of the type of the variable. The returned temporary is of the same type as the variable, as shown in rule C-SET in figure 4.13.

**Cells** A `<make-cell>` creates a cell and emits this as temporary. The type of the cell is the type union of its assignments, as shown in rule C-MAKE-CELL in figure 4.14. A `<get-cell-value>` returns the value of the cell. The value has the type of the cell, as shown in rule C-GET-CELL-VALUE. A `<set-cell-value!>` computation is applicable if a cell of value type $\tau_1$ should get a value of type $\tau_2$, where $\tau_2 <: \tau_1$, as shown in rule C-SET-CV.

$$\frac{\Gamma \vdash_g binding : \tau_1 | C_1 \qquad \Gamma \vdash_g x : \tau_2 | C_2 \qquad \tau_2 <: \tau_1 \qquad C = C_1 \cup C_2}{\Gamma \vdash_g binding := x : \tau_1 | C} \text{ C-SET}$$

Figure 4.13: Constraint generation rules for assignment

$$\frac{\forall_{assignments\ i}\Gamma \vdash_g x_i : \tau_i|\{\} \qquad \tau = \cup\tau_i}{\Gamma \vdash_g make_cell(cell) : cell\tau|\{\}} \text{ C-MAKE-CELL}$$

$$\frac{\Gamma \vdash_g e : cell\tau|C}{\Gamma \vdash_g get_value(e) : \tau|C} \text{ C-GET-CELL-VALUE}$$

$$\frac{\Gamma \vdash_g c : cell\tau_1|C_1 \qquad \Gamma \vdash_g x : \tau_2|C_2 \qquad \tau_2 <: \tau_1 \qquad C = C_1 \cup C_2}{\Gamma \vdash_g set_value(c, x) : \tau_1|C} \text{ C-SET-CV}$$

Figure 4.14: Constraint generation rules for cells

The returned temporary is the new cell value, of type $\tau_1$. These rules are deduced from type rules for references in [ST06, Section 4].

### 4.3.2 Solve Algorithm

The solve algorithm (section 3.3.2) needs to be extended with additional composed types and subtyping. The extended `union` is shown in listing 4.5, `solve` in listing 4.6.

The solution strategy for composed types is propagation to their children. Both types must be of same structure, thus a tuple type cannot be unified with a $\rightarrow$. A special case is variable arity tuple, 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 4.3, 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>`.

The extended relation `order` is changed as shown in listing 4.4. The added lines are 5 and 6, which take care that a rest type (containing variable arity) is not used as representative of any other type, but is preferred a type variable or the dynamic type.

The extended `union` function is shown in listing 4.5. 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, but the more specific value is used as representative. But together with the emitted type checks if the result of an operation returns a broader type then the inferred for a data flow node, all unsafe cases contain run time type checks.

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

Listing 4.3: Type error during type narrowing

$$
\begin{array}{ll}
1 & \texttt{order}(u,v)\ \texttt{=}\ \texttt{case}\ \mathit{type}(u) \simeq \mathit{type}(v)\ \texttt{of} \\
2 & \quad ? \simeq \alpha \Rightarrow (u, v, true) \\
3 & \quad ? \simeq \tau \mid \alpha \simeq \tau \Rightarrow (v, u, true) \\
4 & \quad \tau \simeq \alpha \Rightarrow (u, v, true) \\
5 & \quad \tau \simeq rest \Rightarrow (u, v, true) \\
6 & \quad rest \simeq \tau \Rightarrow (v, u, true) \\
7 & \quad _ \Rightarrow (u, v, false)
\end{array}
$$

Listing 4.4: Extended relation order

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 4.6. 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 $\rightarrow$ constraints.

## 4.4 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 given values. This idea has been proposed in [Kni02].

A result of this specification 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

```
1  union (u, v, order_matters?) =
2    if order_matters? then
3      if u.node − rank = v.node − rank then
4        u.rank := u.rank + 1
5      v.representative := u
6    else
7      if subtype?(u.type, v.type) then
8        u.rank := max(u.rank, v.rank) + 1
9        v.representative := u
10     elseif subtype?(v.type, u.type) then
11       v.rank := max(u.rank, v.rank) + 1
12       u.representative := v
13     elseif u.rank > v.rank then
14       v.representative := u
15     else
16       if u.rank = v.rank then
17         v.rank := v.rank + 1
18       u.representative := v
```

Listing 4.5: Extended Huet union

```
1  case type(u) ≃ type(v) of
2    u1 → u2 ≃ v1 → v2 ⇒ C.add(u1 ≃ v1); C.add(u2 ≃ v2)
3    u1 → u2 ≃ ? ⇒
4      if u.contains_vars then
5        u.contains_vars := false
6        C.add(vertex(type=?, contains_vars=false) ≃ u1)
7        C.add(vertex(type=?, contains_vars=false) ≃ u2)
8    u1 × ... × un ≃ v1 × ... × vn ⇒ C.add(u1 ≃ v1); ... ; C.add(un ≃ vn)
9    u1 ... un ≃? ⇒
10     if u.contains_vars then
11       u.contains_vars := false
12       C.add(vertex(type=?, contains_vars=false) ≃ u1)
13         ⋮
14       C.add(vertex(type=?, contains_vars=false) ≃ un)
15   lcoll_{C_u}(τ_U) ≃ lcoll_{C_V}(τ_V) ⇒ C.add(C_U ≃ C_V); C.add(τ_U ≃ τ_V)
16   τ ≃ var | τ ≃ ? | γ ≃ γ ⇒ pass
17   _ ⇒ error: inconsistent types
```

Listing 4.6: Extended Huet solve

order to avoid this ambiguity 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 mainly used for the instantiation method `make`, which first argument is a type to instantiate, and the return value must be of that type (shown in the results section 5.8.4).

There are still remaining interactions of polymorphic methods and generic functions which are not well specified in this section, and are subject to further work.

# Chapter 5

# Changes to the Compiler and Results

In this chapter I will present the implemented extensions to the compiler in detail. The extensions are support for parametric polymorphism, limited function types and the new type inference algorithm (described in 4.3). The modifications to compiler libraries introduced in 2 are presented. Additionally, a static single assignment conversion was implemented. At the end of this chapter, results gathered from compiling and running the standard Dylan library and its test suite are shown.

## 5.1 Syntax

There are no changes to the parser and lexer. The implemented syntactic extensions are all handled in the definitions library, which is described in the next section.

## 5.2 Enhancing with Semantics: Definitions

The definitions library (section 2.4) is extended in two places: type variable syntax and limited function types syntax.

A method signature may contain a list of type variables. The type variables list is parenthesized and prefixed with the keyword forall::
define method id (forall: A)(x :: A) => (x :: A). The scope of a type variable is the method signature and body. Type variables can be given an upper bound (forall: A <: <list>) where the type variable may only be bound to <list> or subtypes thereof.

Limited function types share the same syntax as a signature, but do not contain names for the bindings, only types. Thus, <integer> => <string> translates to a limited function, which has a single argument of type <integer> and its result is a single value of type <string>. Support for syntax to describe keyword arguments in limited functions remains further work.

## 5.3 Model Objects

The library `dfmc-modeling`, described in section 2.5, is extended with type variable and limited function type classes. Also, `subtype?` and `known-disjoint?` relations are extended to handle these classes, as specified in section 4.2.

## 5.4 Conversion to Flow Graph

The `dfmc-conversion` library, described in section 2.7, is extended with a type variable environment. During conversion, type variables are instantiated and put into the environment.

Also, when a polymorphic method is added to a generic function, the signature is checked for congruency (by treating the type variables as their upper bounds). If a method is added to a polymorphic generic function, the type variables are instantiated and the signature congruency is checked.

An example for congruency checks follow (from the intuition described in section 4.4).

In listing 5.1 the first definition in line 1 is a monomorphic generic function `foo`. When the second definition (lines 3-5) is checked for congruency, the type variables are instantiated and the resulting signature is `(<object>, <object>)`. This signature is not congruent with the signature of the generic function `(<object>, <integer>)`. The last definition (lines 7-9) is congruent with the generic function, because the upper bound of the type variable `T` is an `<integer>`, the resulting signature is `(<integer>, <integer>)`.

The last definition (lines 13-15) is defined on natural numbers. It is rejected because it overlaps with the polymorphic method (lines 7-9). If `foo` is applied to two values which are subtypes of `<integer>`, the polymorphic method is called. If the last definition would not be rejected, application with two positive integer values would be ambiguous. The polymorphic type variable `T` would be bound to the actual argument types, and would clash with the method defined on natural numbers.

## 5.5 Static Single Assignment

In order to estimate a proper type for `a` in the code fragment shown in listing 5.2 (flow graph is in figure 5.1), a representation is needed in which it is possible to have `a` in lines 2-3 to be of type `<integer>`, while the same `a` should be of type `<string>` in lines 4-5. A common solution for this demand is static single assignment form, where every variable is bound exactly once. This is shown in listing 5.3 (flow graph is in figure 5.2). In line 2 `a0` is bound and used in line 3, while in line 4 (the former assignment) a new variable `a1` is bound, which is used in line 5.

Static single assignment is a bit more complicated if branches exist, as shown in listing 5.4 (flow graph in figure 5.3). In this code fragment it is not obvious which `a` to use in line 8. It may be either the one assigned in line 4 or in line 6, depending on the control flow. To manage this problem, so called phi-nodes are emitted, which merge the data flow when the control flow merges, and generate a new data flow element. This is shown in listing 5.5 (line 8) (flow graph in figure 5.4), where a new variable `a3` is introduced (line 8). A phi-node does not

```
1  define generic foo (a, b :: <integer>);
2
3  define method foo (forall: T)(a :: T, b :: T)
4    ⋮
5  end;
6
7  define method foo (forall: T <: <integer>)(a :: T, b :: T)
8    ⋮
9  end;
10
11 define constant <natural> = limited(<integer>, min: 0);
12
13 define method foo (a :: <natural>, b :: <natural>)
14   ⋮
15 end;
```

Listing 5.1: Monomorphic generic function `foo`

```
1  begin
2    let a = 2;
3    let b = a + a;
4    a := "foo";
5    let c = concatenate(a, a);
6  end;
```

Listing 5.2: assignment

```
1  begin
2    let a0 = 2;
3    let b = a0 + a0;
4    let a1 = "foo";
5    let c = concatenate(a1, a1);
6  end;
```

Listing 5.3: SSA-converted assignment

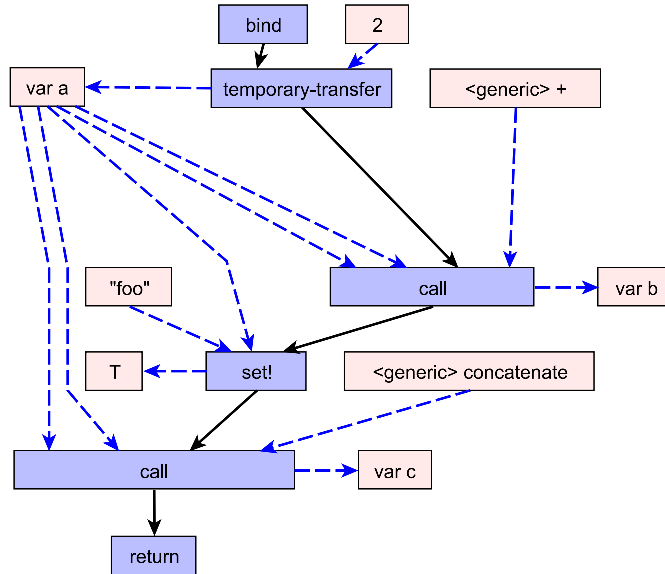


Figure 5.1: Flow graph of a simple assignment

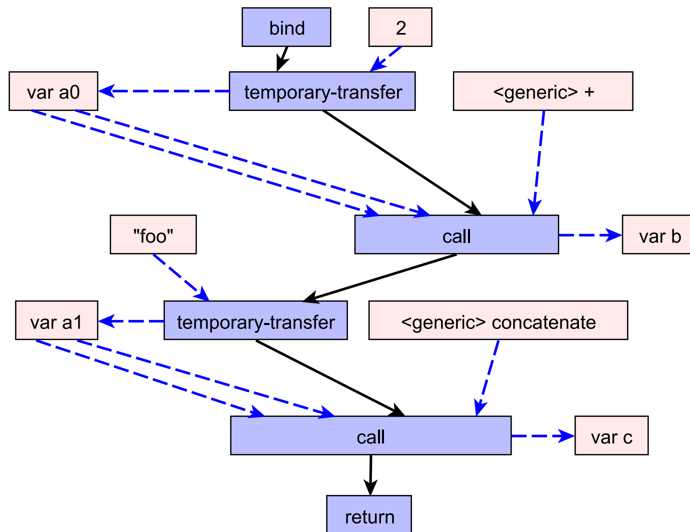


Figure 5.2: Flow graph of SSA-converted assignment

```
1  begin
2    let a = 2;
3    if (b)
4      a := 3;
5    else
6      a := 4;
7    end;
8    a + a
9  end;
```

Listing 5.4: assignment if

```
1  begin
2    let a0 = 2;
3    if (b)
4      let a1 = 3;
5    else
6      let a2 = 4;
7    end;
8    let a3 = phi(a1, a2);
9    a3 + a3
10 end;
```

Listing 5.5: SSA-converted assignment if

exist in a binary; it only exists virtually. A binary cannot decide which value to use, because the information from which branch the code came from is not present in a binary.

The implemented algorithms to convert a flow graph with assignments to a static single assignment form are described in detail in [LEJ79], which computes a dominator tree of a flow graph. A dominator tree contains for every control flow node those nodes which must precede this node, thus which nodes the binary must have visited. With this dominator tree, [CFR+91] is used, to compute placing of phi-nodes in the flow graph, and renaming the users of the original variables to the corresponding dominant data flow nodes.

## 5.6 Compiler Flow

For more precise results, flow graphs are converted into static single assignment form, described in section 5.5. Type inference is done on the static single assignment representation. After initial type inference, the flow graph is translated to a cell representation (described in section 2.6.2), because the optimizer has not yet been changed to use the SSA representation, which remains further work.

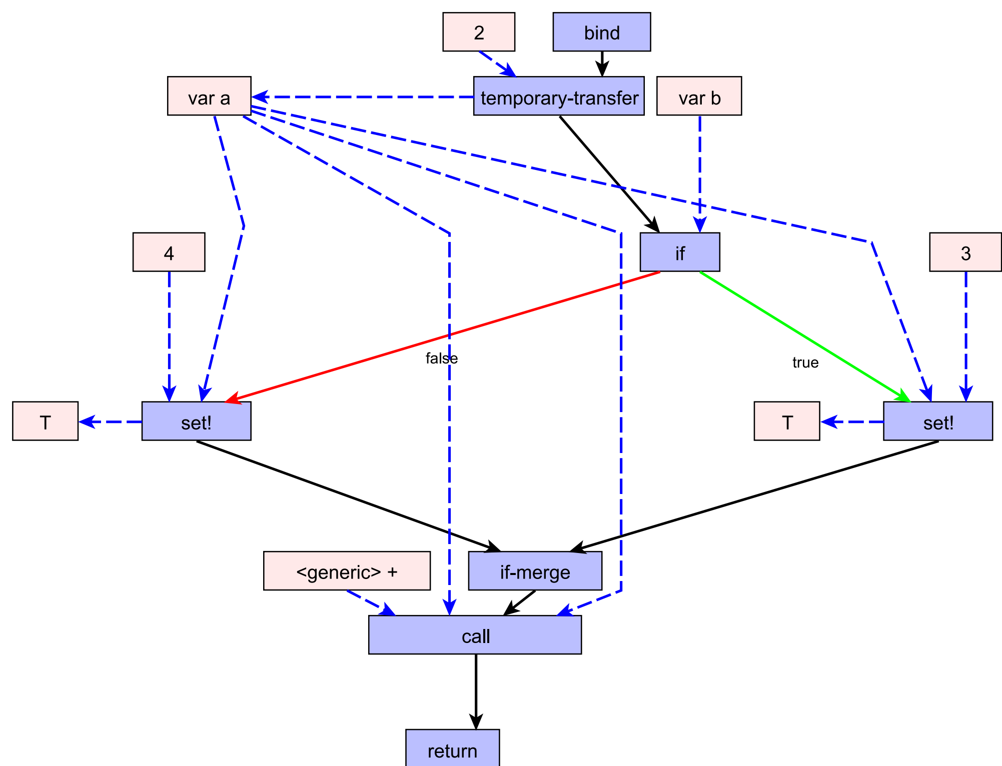


Figure 5.3: Flow graph of conditional with assignment

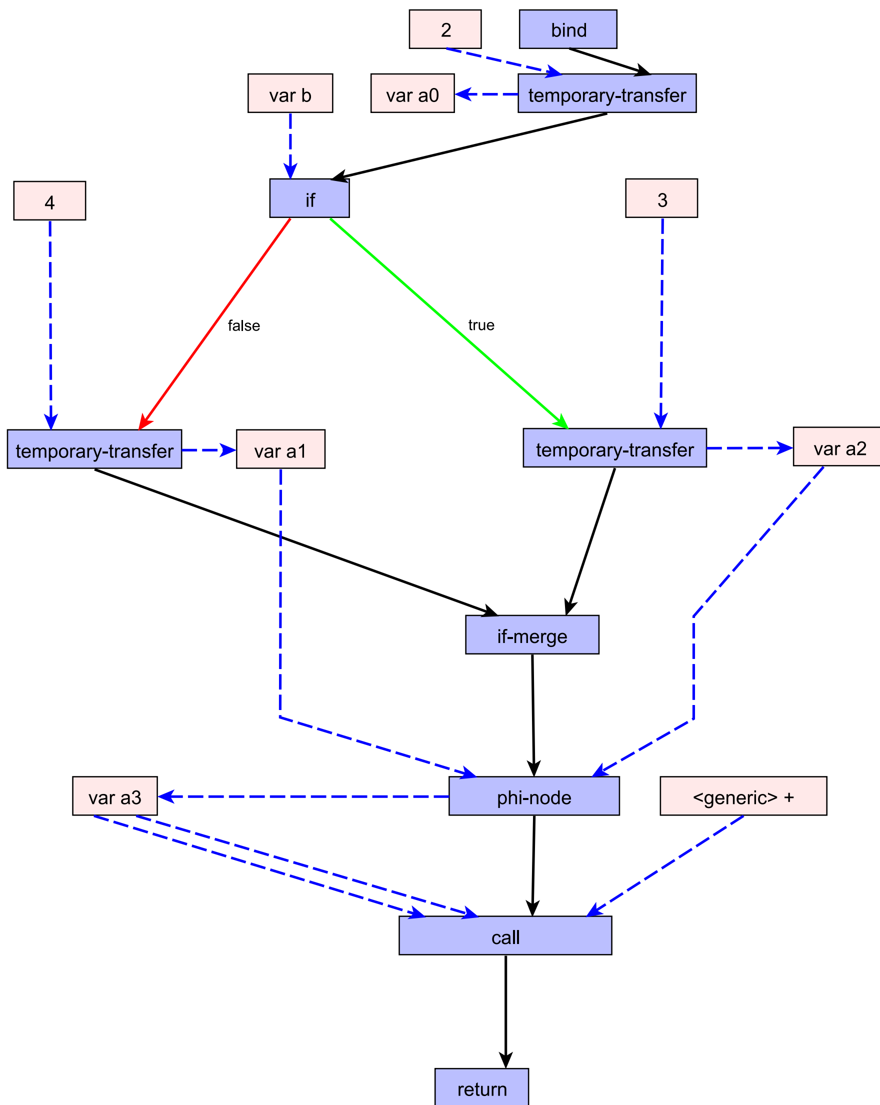


Figure 5.4: Flow graph of SSA-converted conditional with assignment

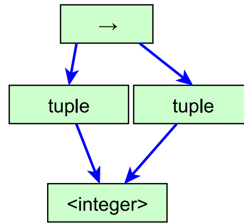


Figure 5.5: Type graph for `<integer>` → `<integer>`

## 5.7 Type Inference

Because the old type inference system (section 2.8) contains several drawbacks and does not support parametric polymorphism, it was rewritten from scratch. It now implements the extended version of the algorithm described in section 4.3.

The inference algorithm is used incrementally, because compiler optimizations like inlining or dead code removal might narrow type estimates. More narrow type estimates might then optimize the flow graph. Whenever a new control flow node is connected into the flow graph, constraints for this node are pushed into the corresponding type graph. When a type estimate of a data flow node is requested, first constraints of the type graph are solved.

In the following, the custom type graph implementation is described, and afterwards type environments are introduced. Finally signature upgrading, an optimization possibility is described.

### 5.7.1 Type Graph

A type graph as described in section 3.3.2 was implemented. Every data flow node in the flow graph has a corresponding type node, which is narrowed as precise as possible. Edges in this type graph can be of three different kinds:

- Graph edges which connect composed nodes, as seen in figure 5.5, where the → is connected to its arguments and values, both a tuple with a single element, `<integer>`.
- Constraint edges reflect an equality constraint between two nodes, as described in section 3.3. The dashed edge from → to ? in figure 5.6 is a constraint edge.
- Representative edges target the representative of a node. An example is shown in figure 5.7, where the dotted edges are representative edges. The topmost ? is represented by `tuple(<integer>)` → `tuple(<integer>)`, the other ? nodes are represented by `<integer>`.

### 5.7.2 Type Environment

In section 3 a code example (listing 3.2) was presented which needs special support by the solve algorithm, namely that ? has to be copied (by `copy_dyn`)

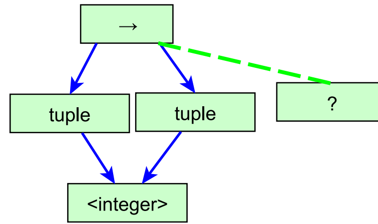


Figure 5.6: Type graph for ? $\simeq$ `<integer>` $\rightarrow$ `<integer>`

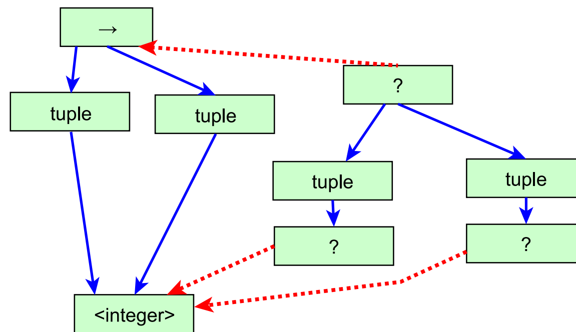


Figure 5.7: Type graph for representative ?: `<integer>` $\rightarrow$ `<integer>`

if used by more than one constraint. In this section another solution for this problem is introduced. The solution are stacked type environments, which is implemented in the Dylan compiler. It has the advantage that the type information for a specific data flow node in a given context is available, and can be narrowed down later (during optimizations).

A new type environment is needed whenever type information of any data flow node diverges, which occurs at control flow branches. The type estimates of the branches would conflict in the same type environment. An example is shown in listing 5.6. The type environment stack ({`TE`}) precedes the actual code. In line 3, the consequence of the conditional of line 2, a new type environment `TE1` is pushed on the stack. In line 5, the alternative of the conditional, the type environment `TE2` is on top of the type environment stack.

In the different type environments, `a` has different types: Type environment `TE0` assigns `a` the $\top$ type. Type environment `TE1` infers `<integer>` as type of `a` because the previous shown method `double` is defined to receive an integer. In `TE2` `a` is of type `<sequence>`, because `concatenate` is defined on sequences. If only a single type environment would be used, `a` would be of type $\top$.

### 5.7.3 Signature Upgrading

After type inference finished, the signature of the inferred top level definition is updated, if it is a subtype. An example is shown in listing 5.7, where no return value is given (resulting in a variable arity tuple type of $\top$ nodes), but

```
1 {TE0}       define method foo (a)
2 {TE0}         if (instance?(a, <integer>))
3 {TE1,TE0}       double(a)
4 {TE0}         elseif (instance?(a, <string>))
5 {TE2,TE0}       concatenate(a, a)
6 {TE0}         end
7 {TE0}       end;
```

Listing 5.6: Type environments

```
define method foo (x :: <integer>)
  x * 2
end
```

Listing 5.7: Signature upgrade opportunity

the resulting value is inferred to be `<integer>`. A check is done whether the inferred return type is a subtype of the return type of the generic function.

### 5.7.4 Example

As simple example again the method `double`, introduced in chapter 2, is used again. The code of method `double` is again shown in listing 5.8, the flow graph in figure 5.8. The initial run of the type inference algorithm builds the type graph shown in figure 5.9.

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 5.10. The inferred type graph is shown in figure 5.11. When this type graph is solved (as shown in figure 5.12), the optimizer can fold the type check, since `MVT1` is guaranteed to contain a single value of type `<integer>`. Analogously to section 2.9.3 the call to `*` is inlined, and the primitives are type inferred. But this does not show any interesting type inference aspects and is omitted here.

## 5.8 Results

In this section several results are presented.

The first result is that the presented algorithm is practically usable in the compiler. The standard Dylan library, containing 32000 lines of code (linecount

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

Listing 5.8: `method double`

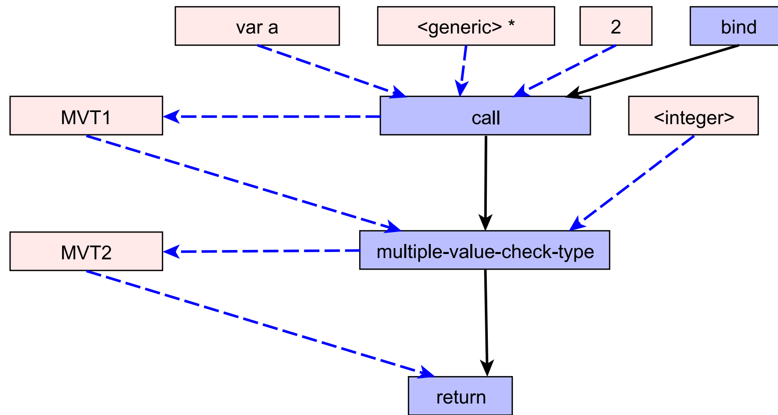


Figure 5.8: Flow graph of method `double`

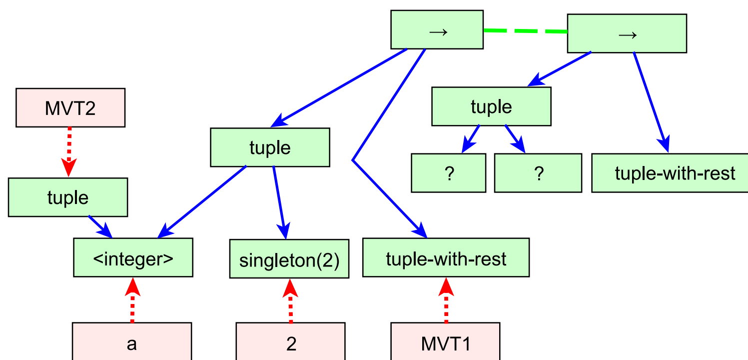


Figure 5.9: Type graph of method `double`

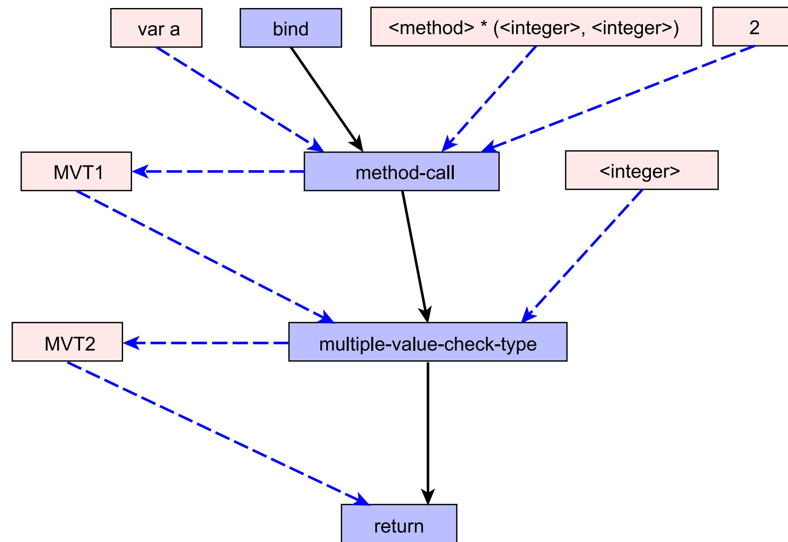


Figure 5.10: Flow graph of upgraded method `double`

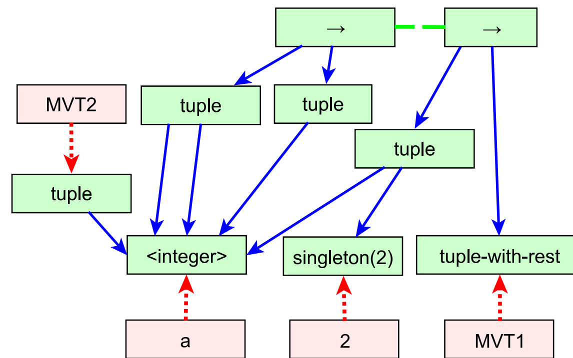


Figure 5.11: Type graph of upgraded method `double`

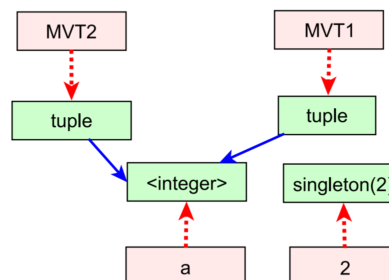


Figure 5.12: Solved type graph of method `double`

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

Listing 5.9: Sample code which is inferred more precise

```
define function map (forall: A, B)
 (A => B, limited(<collection>, of: A))
 => limited(<collection>, of: B)}.
```

Listing 5.10: Polymorphic map

was done with `wc -l`, thus count includes comments and empty lines), compiles with the new type inference algorithm. Only one change was needed for this, namely a type annotation for a result type of a function. The Dylan library test suite containing more than 500 tests and 13000 checks runs with exactly the same results of passes and failures. There is no code coverage tool available, hence I cannot claim correctness of the Dylan library, but the empirical results look promising.

An example which is now type inferred more precise than with the old type inference system is shown in listing 5.9. With the old typist, `i` was inferred to be of type `<object>` and both `<` and `+` were generic function calls. With the new type inference, `i` is inferred to be of type `<integer>` and both calls are direct calls.

Another result is that the described unsafe computation guarantee-type has been removed. Thus, the intermediate representation is safe, a developer is not able to inject unsafe types for bindings.

In the following subsections, the solution to the motivating example is given, a more detailed comparison between the old type inference algorithm (section 2.8) and the new type inference algorithm (section 4.3) is done. Afterwards some problems which have been found in the Dylan library and the DylanWorks compiler are described. And finally some results are presented after the Dylan library was enhanced with polymorphic functions.

### 5.8.1 Polymorphic Functions

Since parametric polymorphic functions are now supported, the motivating code `map(method(x) x + 1 end, #(1, 2, 3))` is shown again. Previously it did a generic function call to `+`, because the type of `map` did not contain a dependency between the collection and the function.

Now it is possible to write a more exact signature for `map`, shown in listing 5.10.

Using the same example as above, `map(method(x) x + 1 end, #(1, 2, 3))`, the literal list has the type `limited(<list>, of: <integer>)`. It constrains the type variable `A` to be an `<integer>`, and the anonymous increment function uses the information that `x` is of type `<integer>`. This anonymous function upgrades the generic function call `+` to a method call of `+(<integer>, <integer>)`.

Then the result value of the anonymous method is upgraded, to be of type `<integer>`. The result type is bound to the type variable `B`, which is upgraded to an `<integer>`. Finally, the return value of the call to `map` is of type `limited(<collection>, of: <integer>)`.

The `map` which was described in this section is a simplified version, the real version in the Dylan library supports a variable number of collections as argument.

### 5.8.2 Comparison to Original Type Inference

An empirical comparison to the old type inference system, described in section 2.8 and in more detail in [Age96], has been done for the Dylan library. The new algorithm uses a SSA-transformed representation, thus the comparison is not a direct comparison between Agesen and Huet. As already mentioned, the Dylan library test suite gives unaltered results.

The results are of three kinds: runtime speed and size during compilation, accuracy of the algorithm, and speed and size of the resulting binary.

The speed decreased roughly by a factor of 3. The size during compilation increased roughly by a factor of 3. This has several causes, the newly implemented type inference is preceeded by a static single assignment transformation. Also type estimates are not cached, during inlining the inlined computations are type estimated from scratch. The speed and size of the type inference algorithm is clearly a topic for further work, it was not the focus of this thesis.

The accuracy of type inference can be measured by the amount of type warnings during compilation. While this is described in detail in the next subsection, the overall result is that the new type inference algorithm is more precise.

The speed and size of the generated binary corresponds primarily to the amount of generic function call and type check nodes in the flow graph. The number of run time generic function calls in the Dylan library increased from 2107 to 2305. The main cause are the described hardcoded types for make and element in the old type inference algorithm (which will be evaluated in more detail in the after next subsection). The number of check type computations increased from 1899 up to 1961.

While the old type inference algorithm was implemented in roughly 5800 lines of code (again, counting was done with `wc -l`, comments and empty lines are counted), the newly implemented algorithm needed fewer than 2600 lines. This is primarily due to the fact that not a completely separate class hierarchy of `<type-estimate>` and utility operations (subtype, disjointness, etc.) was implemented, but the `<type>` class hierarchy was used.

### 5.8.3 Problems Encountered in DylanWorks and the Dylan Library

There were some bugs in primitive declarations, the primitive `primitive-apply` specified a second argument of type `<integer>`, which neither the caller nor the implemented apply was aware of. But, when inferring the signature of the primitive `primitive-apply`, this lead to type mismatches. There were similar bugs in other primitive declarations.

Some other bugs were in the optimizer, where a `primitive-wrap` followed or preceded by a `primitive-unwrap` were optimized because they are reverse

```
⋮
let new :: <profiling-call-site-cache-header-engine-node>
  = ...
primitive-initialize-discriminator(new);
⋮
```

Listing 5.11: compute-terminal-engine-node

```
⋮
let m :: <method> = head(subordered);
let more
  = if (function-next?(m))
⋮
```

Listing 5.12: transmogrify-method-list-tail-grounded

operations. This is true, while the former adds tag bits, the latter removes them. But on a type level, any raw value can be unwrapped to a machine word and then wrapped to any other raw value, thus raw-addresses were unwrapped to machine words and wrapped into raw-integers. The type inference complains if a raw-address is given where a raw-integer is expected.

Another optimization was to upgrade a loop variable which had the type specification integer directly into a machine-word. This optimization may safe wraps and unwraps of the loop variable. But the specification of the loop variable still was integer. When the loop got inlined somewhere else, the type inference could not unify the type of the loop variable, being a machine word and an integer at the same time.

In the following, some type errors are shown, which were not found by the old type inference algorithm. In listing 5.11, a new engine node object of the specified type is allocated. The engine node object is then passed to the initialization method `primitive-initialize-discriminator`, which is specified for `<discriminator>` instances. Engine nodes and discriminators are disjoint. This results in a run time type error, which was not encountered because profiling was disabled globally.

Another error is shown in listing 5.12, where `function-next?` is specified on `<lambda>`, and the argument is a `<method>` instance. While `<lambda>` is a subclass of `<method>`, there also exists `<accessor-method>`. This would also result in a run time type error, but did not for unknown reason, either because the code was never called or never with an accessor method.

### 5.8.4 Comparison to Polymorphic Dylan Library

With the original type language developers were not able to express that the generic function `make` (original signature in listing 5.13) contains a dependency between the given type and the returned instance. This dependency is specified in the documentation [Sha96, Chapter 12]: “The instance returned is guaranteed

```
define generic make
 (type :: <type>, #rest, #key, #all-keys)
 => (instance :: <object>)
```

Listing 5.13: Original signature of `make`

```
define generic make
 (forall: type)
 (type :: <type>, #rest, #key, #all-keys)
 => (instance :: type)
```

Listing 5.14: New signature of `make`

to be a general instance of type but not necessarily a direct instance of type.“

The new type expression language allows the signature to be specified as shown in listing 5.14. It contains the type variable `type`, which is bound to the value of the first argument, which is a type, either a `singleton` or a `subclass`. The return value has to be an instance of the given type. This found several inconsistencies of the definition of methods for make, which had no type specification of the result value. The signature is checked when the defined method is added to the generic function, thus before type inference and signature upgrading takes place.

Analogously this applies to the conversion method `as`, which takes a type and an object and converts the object to the type. This improved the generated code at several places, namely where `as` was called, but could not be upgraded to a method call, and the result was used later. Now, even if the generic function `as` is called, the result value will be of the specified type. All methods of `as` are verified to fulfill this contract. An example is the `element-setter` of a string containing bytes, shown in listing 5.15. This method first converts the received character into a byte character, and then replaces the character at the given index with the new byte character. The element setter call was previously a generic function call, because the result value of `as` was an object, and now it is a direct repeated slot access.

After changing both `make` and `as`, the amount of run time generic function calls dropped from 2305 (new type inference without polymorphic methods) down to 2093. Thus, by only changing two generic function signatures to polymorphic methods, the number of generic function dispatches decreased by more

```
define inline sealed method element-setter
    (new-value :: <character>,
     string :: <byte-string>,
     index :: <integer>)
 => (character :: <byte-character>)
  string[index] := as(<byte-character>, new-value);
end method element-setter;
```

Listing 5.15: Method `element-setter`

than 100. Compared to the old type inference algorithm, the number of generic function calls decreased by 14. This is similar for the number of type check computations, there are now 1882 present, which are 79 fewer than without polymorphic methods. It is also by 17 lower than the number of type checks which were emitted by the old type inference algorithm. Thus, the presented and implemented algorithm leads to fewer type checks and fewer generic function dispatches.

Further work also consists of rewriting the collection code to use polymorphic functions, which needs support of variable arity polymorphism. This will lower the number of generic function calls, because currently collections, if not manually specified to be of a special element type, are not inferred to contain a specific element type. Also, operations on collections still contain no dependency between the given function and argument collection.

# Chapter 6

# Related Work

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 [CF91]. 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.

A more recent approach is typed Scheme [THF08], which is a type system for PLT Scheme. It integrates type annotations into Scheme and polymorphic type variables. The unique feature of their type system is occurence typing, which is composed of visible predicates and latent predicates. Visible predicates contain information about flow control information. Latent predicates are types atop the $\rightarrow$ type which include type information of the results, so a predicate *number*? does not only have the type $Object \rightarrow Boolean$, but also include the information that if the result is true, the argument will be of type $Number$, written $Object \rightarrow^{Number} Boolean$. Scheme code can quite easily be ported to typed Scheme code and it is also possible to use typed and untyped modules side by side, while the typed ones cannot be blamed [WF09]. The granularity of typed Scheme is on a module basis, while in gradual typing it is a finer level of granularity. Run time type errors cannot originate from safe regions.

Another approach is hybrid type checking [GKT+06], which contains a Dynamic type for the unknown values. The type system supports type refinements, which are constraints for types, like limited types in Dylan. But the type refinement approach is more powerful. Additionally types are first-class values (types can be returned from functions, and it can be abstracted over types). When a value has a type but another type is expected, and subtyping cannot be computed at compile time, a downcast (run time type check) is inserted. Also, type checking is deferred to run time if the subtype algorithm takes too much time. A notable difference to my approach is that refinement types may

contain any predicate, which are solved with a theorem prover.

Formalization of type safety and multiple dispatch was done in Cecil [Cha92], more specific in [LC94]. Cecil has a prototype-based object system while Dylan has a class-based object system.

Kea [MHH90] is a statically typed programing language which integrated multi-methods and polymorphism. Kea also preserves encapsulation, methods may still be organised within classes. As a result the first argument is still given a special status.

Integration of structural and nominal type systems was also studied in Unity [MA08].

The blame calculus [WF09] provides a high-level semantics of casts and tracing of errors back to the source. This has also been incorporated into gradual typing in [SW09], where the relationship between low-level casts and high-level casts has been characterized precisely. The main idea is to summarize a sequence of casts as a threesome containing source type, target type and a middle type, the greatest lower bound of all types in the sequence. This work looks promising, and should be integrated into the presented solution as further work.

An approach which integrates static types into Ruby is [FhDAFH09]. They came up with a type system for Ruby, a type annotation syntax and a type inference algorithm. 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 run time method redefinition. In Dylan, it is specified that only non-sealed methods might be replaced at run time, thus such a restriction is not present in the described system. 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. These false positives can be prevented by the integration of occurence typing, as described in the typed Scheme paragraph.

Integration of parametrized types into GOO, a dynamically typed programming language focussing on real time audio and video transformation, was done in [Kni02]. 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 this thesis. In the thesis function types cannot be parametrized, which is the main work of this thesis.

# Chapter 7

# Conclusion and Further Work

In this thesis I have shown the practicability of the implementation of a type inference algorithm based on a formal type system into an existing dynamically typed programming language. The results look promising: when parametric polymorphism is used, the implemented type inference algorithm is more precise and enables more optimizations compared to the original implemented algorithm. The described algorithm reports more type errors at compile time and the generated code contains fewer computations which may fail at run time, namely type assertions and generic function calls.

The running speed of the implemented algorithm is currently slower than of the original algorithm. This has two main causes: on the one hand type information is not yet cached, on the other hand the control flow graph is first converted into a static single assignment form.

Additionally the implemented extensions (function types and parametric polymorphism) for the Dylan type syntax enable developers to specify more narrow types for bindings. This makes the type system of Dylan more powerful and leads to higher performance and fewer type errors at run time.

Also, the animated graph visualization 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 ([Pie02]) and the gradual typing system ([ST06], [SV08], [ST07]), but there are no formal proofs of the combined type system.

Since the static single assignment conversion has been implemented, the optimizer should use this representation instead of the cell representation. This would enable more optimization possibilities. After the optimization phase finished, the resulting flow graph must be converted to a cell representation which is used by the backends. During cross library optimization, also the cell representation should be used.

Also, Dylan's list comprehension is defined on variable number of arguments, the recently published paper covering variable arity polymorphism [STHF09]

will be integrated. This was developed by the typed Scheme research group, Scheme also contains variable arity list comprehension functions.

Afterwards the collection implementation of Dylan can be rewritten with polymorphism. This will on the one hand decrease code size (lines of source code), since limited collections need no further special support. On the other hand, it will improve type inference and thus minimize generic function calls and type checks. Also, the hash table implementation currently makes no use of polymorphism, but it should be aware that it contains keys of type $\alpha$ and values of type $\beta$, then getting and setting elements and keys need no generic function calls, as well as rehashing of the table. All three operations currently involve generic function dispatches.

It would be useful to cache polymorphic methods which have been upgraded to a monomorphic method, because copying the method body every time and optimizing it for identical types is duplicated work. An issue is where to cache those methods, since a library `foo` might define a polymorphic function, and library `bar` and `boo` might use the polymorphic function, but do not know of each other. Neither `bar` nor `boo` can modify the binary or metadata of `foo`.

An additional feature for Dylan's type system will be occurence typing [THF08], 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 and also composed predicates can be used for type information.

# Bibliography


[AC] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer.

[Age95] Ole Agesen. The cartesian product algorithm. In *Proceedings of ECOOP 1995*. ACM, 1995.

[Age96] Ole Agesen. Concrete type inference: Delivering object-oriented applications. Technical report, Mountain View, CA, USA, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BP99] Peter Buneman and Benjamin Pierce. Union types for semistructured data. Technical report, University of Pennsylvania Dept. of CIS, 1999.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[Cha92] Craig Chambers. Object-oriented multi-methods in cecil. In *In ECOOP '92 Conference Proceedings*, pages 33–56. Springer-Verlag, 1992.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: constraining parametric polymorphism. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 156–168, New York, NY, USA, 1995. ACM.

[FhDAFH09] Michael Furr, Jong hoon (David) An, Jeffrey S. Foster, and Michael W. Hicks. Static type inference for ruby. In *SAC*, pages 1859–1866, 2009.

[GKT+06] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *In Workshop on Scheme and Functional Programming*, 2006.

[Hue76] G. Huet. Resolution d'equations dans les langages d'ordre 1, 2, ..., omega. 1976.

[Kni02] James Knight. Parametrized types for goo. 2002.

[LC94] Gary T. Leavens and Craig Chambers. Typechecking and modules for multi-methods. *ACM Transactions on Programming Languages and Systems*, 17:1–15, 1994.

[LEJ79] Thomas Lengauer, Robert Endre, and Tar Jan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1:121–141, 1979.

[MA08] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 260–284, Berlin, Heidelberg, 2008. Springer-Verlag.

[MB95] Tony Mann and Jonathan Bachrach. Harlequin dylan runtime system. 1995.

[MHH90] W. B. Mugbridge, J. G. Hosking, and J. Hamer. Functional extensions to an object-oriented programming language. Technical report, 1990.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[SH06] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Trans. Comput. Logic*, 7(4):676–722, 2006.

[Sha96] Andrew Shalit. *The Dylan Reference Manual*. Addison Wesley, 1996.

[Shi05] Olin Shivers. The anatomy of a loop: a story of scope and control. In *ICFP*, pages 2–14, 2005.

[ST06] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of Scheme and Functional Programming 2006*. ACM, 2006.

[ST07] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *Proceedings of ECOOP 2007*. ACM, 2007.

[STHF09] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *ESOP '09: Proceedings of the Eighteenth European Symposium On Programming*, pages 32–46, March 2009.

[SV08] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of PLDI 2008*. ACM, 2008.

[SW09] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *STOP '09: Proceedings for the 1st workshop on Script to Program Evolution*, pages 34–46, New York, NY, USA, 2009. ACM.

[THF08] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of POPL 2008*. ACM, 2008.

[WF09] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag.

# Anhang A

# Zusammenfassung

Features von statisch typisierten Programmiersprachen sind gut und nützlich, sie fehlen jedoch bisher in dynamischen Sprachen. Die Features von statisch typisierten Sprachen sind sowohl frühe Fehlererkennung (Typfehler) als auch Optimierungsmöglichkeiten.

In dynamischen Programmiersprachen sind Typ-Systeme bisher nicht formal erfasst und aus diesem Grund nicht automatisch verifizierbar, auch die Korrektheit der Typ-Checking Algorithmen wurde bisher nicht formal bewiesen.

Während einige theoretische Betrachtungen existieren, legte ich darauf wert, meine Erkenntnisse sofort praktisch visualisieren und überprüfen zu können. Ferner gab mir die Orientierung an einer real eingesetzten Programmiersprache die Möglichkeit, Abschätzungen über die Praktikabilität einer Integration eines Typ-Systems in existierende Software zu machen. Ich habe ein ganzes Kapitel der Arbeit der praktischen Implementierung des Typ-Systems gewidmet.

Der Text ist für Wissenschaftler und Programmierer mit Interesse an Typ-Systemen und dynamischen Programmiersprachen gedacht. Auch Programmiersprachen und Compiler-Entwickler in anderen als der beschriebenen Programmiersprache Dylan können von den gemachten Erfahrungen profitieren. Durch die formale Grundlage wurden Fehler im Programmcode auch des Dylan-Compilers zu Tage gebracht, analoge Implementierungen unter Berücksichtigung der in dieser Arbeit vorgestellten Ergebnisse können auch in anderen Sprachen zu einer Verbesserung der Softwarequalität führen.

Programmierern werden Werkzeuge in die Hand gegeben, eine präzisere statische Analyse (für Fehlererkennung und Optimierung) während des Compilens durchzuführen.

Während der Forschung zur Arbeit habe ich eine anschauliche Visualisierung des Programm- und Datenflusses als ein mächtiges Werkzeug zum Nachvollziehen von Arbeitsabläufen und Erkennen und Vermeiden von Hindernissen für effiziente Code-Erzeugung entwickelt. Das Visualisierungs-Werkzeug hilft dabei, zu verstehen, warum der Compiler die Typ-Korrektheit bestimmter Ausdrücke nicht nachvollziehen und Optimierungen nicht durchführen kann. Dies kann auch anderen Programmiersprachen-Entwicklern bei ihrer Arbeit helfen.

Neben der Visualisierung des Kontroll- und Datenfluss werden auch die Arbeitsschritte und Zustände des Typ-Inferenz-Algorithmus dargestellt, dieses gibt zukünftigen Entwicklern ein einfaches aber mächtiges Tool zur Fehlersuche und zum Nachvollziehen von Missverständnissen zwischen Programmierer und Com-

piler.

Abschließend untersuche ich den praktischen Gewinn, der durch die im Rahmen der Arbeit implementierten Features erzielt wurde anhand einfacher Metriken wie der Anzahl von generic function calls und Typenüberprüfungen zur Laufzeit. Ich stelle Fehler vor, die durch die neu gewonnenen Möglichkeiten des Typ-Systems in existierender Software, der Dylan Standardbibliothek und im Compiler, entdeckt wurden.