TYPE-BASED ALIASING CONTROL FOR THE DISCIPLINED DISCIPLE COMPILER

Guiding Compiler Optimisation with Aliasing Information
in a Type System with Regions and Effects

Author

Tran Ma

Supervisor

Ben Lippmeier

Assessor

Manuel Chakravarty

A thesis submitted for the degree of
BSc. Computer Science (Honours)

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

October 2012

Abstract

Aliasing is an important problem in compiler optimisation, traditionally resolved with static analysis. We present a type-based approach to the problem that focuses instead on aliasing control and show preliminary evidence of its practical merit.

Our work relies on the type system of Disciple, a strict functional language with support for type-safe impure operations. The first part of this report deals with the necessary extensions to Disciple to enable the tracking of aliasing information. The second half concerns with marshalling aliasing data from Disciple's type system to a format recognisable by LLVM, our compiler back-end of choice. Ultimately our goal is to demonstrate the potential of the type discipline in Disciple with regard to program optimisation.

Thanks

I would like to thank my advisor, Ben, for being very helpful and patient, and also for writing the best Haskell code base I have ever seen.

Thanks to Liam for providing useful insights into the problems I had when writing this thesis, and also everyone else in the PLS research group for their input during the course of the year.

Thanks to my employers at UNSW and NICTA for helping me cope with the stress of this busy year.

Finally I would like to thank my parents for their support in everything.

Contents

1 Introduction

1.1 Motivation and Goals

One of the novel contributions of Disciple and DDC [Lip10] was a system with region, effect and closure typing that is capable of expressing properties of store locations in the type system. Knowledge of these properties plays a critical part in compiler optimisation, and much research has been done on the topic of analysing programs to recover this information. In particular, many techniques were developed to determine the aliasing data of programs. With Disciple, such information can be encoded in the type system rather than discovered by post-hoc analysis. Our aim is to realise this potential and evaluate the effectiveness of this approach with regard to optimisation opportunities. To this end, we lay out our most critical goals:

- Implement a way to represent aliasing information with Disciple's type system.
- Devise and implement a conversion from this representation to a format recognisable by DDC's LLVM back-end.
- Evaluate the effect of using Disciple's aliasing data to guide aliasing-based optimisations in LLVM.

1.2 Overview

This thesis is organised into the following parts:

Introduction presents the goals and structure of this report.

Background discusses the necessary background information of our work, including a brief overview of Disciple's type system and language, as well as a description of some relevant features in LLVM.

Control Aliasing with Witnesses presents the extensions to Disciple's type system to facilitate tracking of aliasing information.

Metadata Generation covers the design and implementation of the mechanism by which we transfer aliasing information in DDC to LLVM.

Optimisation Results evaluates the observable results of our approach.

Conclusions discusses some related work as well as possible improvements and extensions to our work.

2 Background

This chapter gives a brief overview on the systems involved in our work: the Disciple language, the architecture of the Disciplined Disciple Compiler (DDC) and the LLVM framework (formerly Low-Level Virtual Machine). Additionally, we discuss in more detail the aliasing problem with regard to optimisation, as well as solutions in related systems.

2.1 Type System of Disciple

We briefly introduce the type system of Disciple, originally descirbed in [Lip09] and [Lip10], focusing on regions and witnesses.

2.1.1 Regions, Effects and Closures

Disciple uses region types to allow for mutability polymorphism; a *region variable* in Disciple is a name for an abstract set of store locations, much like in [TJ92]. Region variables exist on the type level and thus can be used to parametrise other type constructors. For example, consider the kind of *Int*:

```
Int :: % ⇝ *
```

Where *Int* is a type constructor that takes a region kind (%) and produce an integer. Note that Disciple differentiates syntactically between the type of value-level functions (→) and the kind of type-level functions (⇝).

Destructive update requires a value to be mutable; a property of the store that can be ensured with the use of *region constraints*:

$$\texttt{updateInt :: [r1 r2 : \%]. Mutable r1} \Rightarrow \texttt{Int r1} \rightarrow \texttt{Int r2} \xrightarrow{Read\ r_2 + \ Write\ r_1\ \mid\ Use\ r_1} \texttt{Unit}$$

Here, $[r_1\ r_2\ :\ \%]$ is a type abstraction and *Mutable* r_1 is a constraint that guarantees the mutability of the first argument. An analogy for region constraints is type classes in Haskell, due to the similar fashion in which they are resolved. Where a type class, say *Eq*, can be explicitly reified in GHC by passing a dictionary to the function that requires it, the region class *Mutable* is satisfied by passing in a *witness* term that serves as evidence for mutability.

The last function arrow in *updateInt* is annotated with the *effect* types *Read* r_2 and *Write* r_1, which are the observable actions that *updateInt* performs on r_1 and r_2. *Read* and *Write* are effect constructors that produce an effect on a particular region:

```
Read :: % ⇝ !
```

Effects are collected with the + operator. Pure functions have the effect $\bot$; unannotated function arrows → are simply sugar for $\xrightarrow{\bot}$, thus partially applying the first argument to *updateInt* is considered pure.

Besides effects, functions in Disciple are also annotated with closure types to express the sharing properties of regions [Lip10]. In the case of *updateInt*, this closure is *Use* r_1, which indicates that r_1 is captured in the closure.

2.1.2 Witnesses

Reification

Region constraints in Disciple are resolved in a similar manner to type classes in Haskell. Consider the following Haskell function that tests if a value exists in a list:

```
member :: Eq a ⇒ a → [a] → Bool
```

The type constraint *Eq* is resolved at compile-time by requiring a dictionary that contains the appropriate functions to perform the comparison as an argument to *member* [WB89]. For example, the application *member 1* $[\mathit{1}, \mathit{2}, \mathit{3}]$ yields something similar to *member dictEqInt 1* $[\mathit{1}, \mathit{2}, \mathit{3}]$, where *dictEqInt* contains the primitive equality function on integers (i.e. $\mathit{dictEqInt} = \langle \mathit{primIntEq} \rangle$).

Similarly, recall that *updateInt* asks for a mutable argument:

```
updateInt :: [r1 r2 : %]. Mutable r1 ⇒ Int r1 → Int r2 → ()
```

An application of *updateInt* requires some evidence that the first argument, say x_1, is mutable; we call this evidence a witness of mutability for x_1. Assuming the witness already exists, it is explicitly passed in to the function call: *updateInt* $\langle w_1 \rangle$ x_1 x_2. Witnesses in Disciple are on the same level as value terms, but they are separate entities, so we use the special syntax $\langle w_1 \rangle$ to differentiate.

Construction

We now examine how witnesses are constructed. If a witness is constructed at the call site, e.g.

```
updateInt MkMutable x1 x2
```

The compiler would have to check the whole program to ensure that incompatible constraints do not exist for the same region, e.g. *Const r* and *Mutable r*. Instead Disciple only allows witnesses to be constructed at the same time as the region variables to which they refer [Lip10]. This is done with a **letregion** construct, similar to MLKit's approach [BLE$^+$06].

The term **letregion** r **with** $\{\overline{w_i : \delta_i}\}$ **in** t binds a region variable r (in scope in both the witness bindings $\{\overline{w_i : \delta_i}\}$ and the body t). The reduction of this expression introduces a fresh region handle and substitutes it for r in t, as well as setting the properties of the new region. Returning to our running example, we construct witnesses and use them on *updateInt* as follows:

```
letregion r1 with {w1 : Mutable r1} in
letregion r2 with {w2 : Const   r2} in
  do x = 5 [r1] ()
     updateInt [r1] [r2] <w1> x (42 [r2] ())
```

Where the syntax $[\tau]$ refers to the application of the type argument τ. In Disciple literals are simply functions that take a region handle and a unit value to produce a value in that region. For example, the type of the literal 42 is:

$$\texttt{42 :: [r : \%]. Unit} \xrightarrow{\mathit{Alloc}\ r\ \mid\ \$0} \texttt{Int r}$$

Here *Alloc* is the effect type for memory allocation and \$0 is the empty closure. In the code example above, 42 [r_2] () allocates an integer in the heap region given by r_2, sets its value to 42 and returns the integer object.

Restricting the introduction of witnesses to the **letregion** construct ensures that conflicting witnesses cannot be created, simply by imposing some syntactic constraints [Lip09].

- **Uniqueness:** If a **letregion** binds a variable r, only r can be used in the witness bindings.
- **Well-formedness of region witnesses:** The binders $\overline{w_i : \delta_i}$ cannot contain witnesses of types *Const* and *Mutable* for the same w_i.

2.2 Disciplined Disciple Compiler (DDC)

2.2.1 Overview

Disciple has been undergoing heavy development over the past few years, so the current system is substantially different from the instances in published literature [Lip10, Lip09]. We present an overview of the current snapshot of the compiler and its languages, mostly based on the descriptions in [Lip12] and the development state.

DDC can be thought of as two compilers for two closely related languages [Lip12]:

- The *Disciple Source Language* is the actual language in which Disciple programs are usually written. Type inference is done on this language to produce the more verbose but simpler to check Core IR.
- The *Disciple Core Language* is based on System-F_c with the region and effect system of [TJ92] combined with closure typing [LW91]. The language is call-by-value by default, but lazy evaluation is supported for functions with no visible side effects.

Our work focuses on the back-end part, where Core IR is compiled to machine code via either C or LLVM:

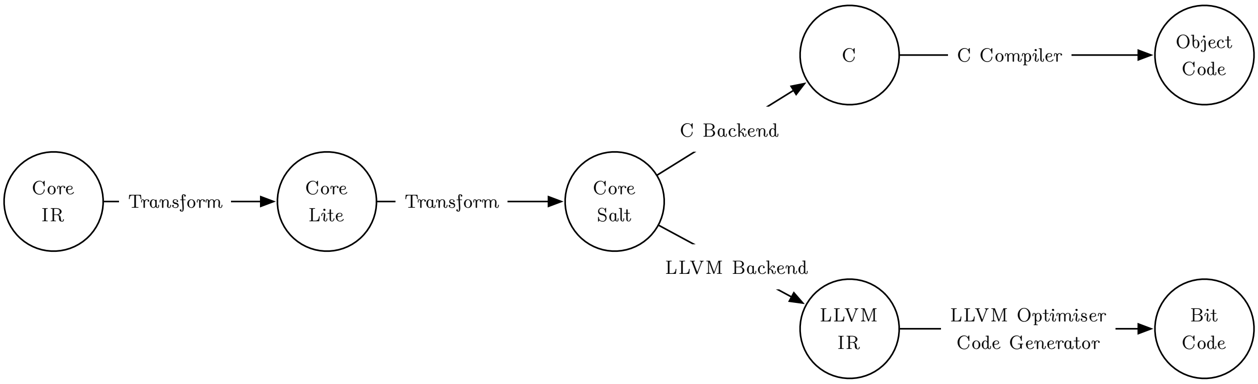


Figure 2.1: Compilation Phases for Core IR

Lite and Salt are language fragments of Core. They use the same AST but have restricted sets of features. The important points about these two fragments are:

- Lite has System-F style parametric polymorphism and support for data types.

- Salt is a minimal language that can be easily translated to C or LLVM IR. It has functions, case expressions and a set of primitive operations, including C style pointer arithmetic. In the Lite to Salt transformation, operations on data types are expanded into these operations.

2.2.2 Core IR

Universes

There are four levels to the universe hierarchy in Disciple Core:

- Level 0: *Data values* and *witnesses* exist separately on this level; the former is analogous to the *Set* universe in the calculus of constructions (e.g. the logic in Coq), while the latter can be thought of as the *Prop* universe. Data values exist at runtime, whereas a witness is a proof object for a property of an effect or region, used by the compiler and erased before runtime. Example terms at this level are $(\lambda\ (x : \mathit{Unit}).\ x)$ and $(\mathit{read}\ [r_1]\ \langle w_1 \rangle)$.
- Level 1: *Specifications* are equivalent to types in Haskell, but the term "type" is overloaded in Disciple to refer to kinds and sorts also. A specification of a computation gives an upper bound of the run-time effect, e.g. for the data example above, the specification is $\mathit{Unit} \rightarrow \mathit{Unit}$. A specification of a witness describes the property that the witness proves by its construction, e.g. $\mathit{Mutable}\ r_1$ or $\mathit{Pure}\ (\mathit{Read}\ r_1)$.
- Level 2: *Kinds* classify specifications, e.g % for region variables.
- Level 3: *Sorts* classify kinds.

Variables and Constructors

Variables can be named or indexed with de Bruijin indices. Constructors can be textually or symbolically named; by convention constructors ending in a hash are used for unboxed types, e.g. `42i#` is a data constructor for `Int#`, Core Salt's primitive integer type.

$$\begin{array}{lcl}
\mathbf{var} & ::= & \mathbf{namedvar} \mid \mathbf{indexvar} \\
\mathbf{namedvar} & ::= & \text{lower } \mathbf{varchar}^* \\
\mathbf{indexvar} & ::= & \text{hat digit}^+ \\
\mathbf{varchar} & ::= & \text{lower} \mid \text{upper} \mid \text{digit} \mid \text{underscore} \mid \text{prime} \\
 & & \\
\mathbf{con} & ::= & \mathbf{namedcon} \mid \mathbf{symbolcon} \\
\mathbf{namedcon} & ::= & \text{upper } \mathbf{varchar}^* \ \#? \\
\mathbf{symbolcon} & ::= & \mathtt{@@} \mid \mathtt{**} \mid \mathtt{*} \mid \mathtt{\%} \mid \mathtt{!} \mid \mathtt{\$} \mid \mathtt{@} \mid \mathtt{(\sim>)} \mid \mathtt{(=>)} \mid \mathtt{->}
\end{array}$$

Types

Specifications, kinds and sorts use the same grammar:

$$\begin{array}{rcl}
\mathbf{type} & ::= & \mathbf{var} \\
 & \mid & \mathbf{con} \\
\text{(type application)} & \mid & \mathbf{type}\ \mathbf{type} \\
\text{(universally-quantified type)} & \mid & \mathtt{[}\ \mathbf{binder} : \mathbf{type}\ \mathtt{]}\ \ \mathtt{.}\ \mathbf{type} \\
\text{(type sums)} & & \\
 & \mid & \mathbf{type} + \mathbf{type} \\
 & \mid & \mathtt{!0} \\
 & \mid & \mathtt{\$0}
\end{array}$$

With the constructor *Int* and some type variable r of kind %, we can construct the type application *Int* r; furthermore we can universally quantify this type with $[r : \%]$. *Int* r. Here r is a named binder, but in general binders can be named, indexed with De Bruijin indices (e.g. [^ ^ : %]. *Int* ^1 → *Int*^0) or entirely anonymous (e.g. [_ : %]. *Unit*).

$$\textbf{binder} \quad ::= \quad \textbf{namedvar} \mid \textbf{hat} \mid \textbf{underscore}$$

Type sums represent the least upper-bound of its two arguments, which can be types, effects or closures. The constructors $0 and !0 are the empty closure and empty effect respectively.

Witnesses

A witness can be a witness variable, a witness constructor, application or join:

$$\begin{array}{rcl}
\textbf{wit} & ::= & \textbf{var} \\
 & ::= & \textbf{wcon} \\
\text{(application)} & ::= & \textbf{wit warg} \\
\text{(join)} & ::= & \textbf{wit}\ \&\ \textbf{wit} \\
 & & \\
\textbf{warg} & ::= & \textbf{wit} \mid [\ \textbf{type}\ ]
\end{array}$$

As discussed in section 2.1.1, witnesses can only be constructed as a binding in a **letregion**, so **wcon** here refers only to built-in constructors that encodes axioms in the type system.

At runtime the witnesses introduced with **letregion** are replaced with store capabilities represented by the witness constructors:

```
Global#  :: [r : %]. Global r
Const#   :: [r : %]. Const r
Mutable# :: [r : %]. Mutable r
Lazy#    :: [r : %]. Lazy r
Direct#  :: [r : %]. Direct r
```

When applied to a region variable, say r_1, *Const#* produces a witness of type *Const* r_1. Details on other witness constructors, including built-in ones, are given in [Lip12].

Expressions

Expressions in Disciple include all the typical forms for an ML-like language, such as case expressions and let bindings. The distinguishing features of Disciple Core are region constructs and type casts:

```
                                     exp  ::=  var
                                            |  con
                                            |  literal
                              (applications)
                                            |  exp xarg
                       (lambda abstractions)
                                            |  Λ( binder : type ) . exp
                                            |  λ( binder : type ) . exp
  (non-recursive and recursive let binding)
                                            |  let letbind in exp
                                            |  let letbind lazy < witness > in exp
                                            |  letrec { letbindtyped;⁺ } in exp
        (region introduction and context)
                                            |  letregion binder with { typesig;* } in exp
                                            |  withregion var in exp
                         (case expression)
                                            |  case exp of { alt;⁺ }
                               (type casts)
                                            |  weakeff [ type ] in exp
                                            |  weakclo [ type ] in exp
                                            |  purify < witness > in exp
                                            |  forget < witness > in exp
```

We saw in section 2.1 how the **letregion** construct is used to bind a region variable together with its properties. The other region handling construct, **withregion**, is used to hold a region handle during evaluation.

Details on type casts and other expressions are given in [Lip12].

2.3 Aliasing

2.3.1 Aliasing and Optimisation

Aliasing refers to the scenario where a storage location can be accessed via different ways. The aliasing set of a variable refers to the set of all variables it may alias. The goal of alias analysis is to determine the minimum aliasing set for all variables in the program so that optimisations can be as aggressive as possible. Many optimisation methods rely heavily on alias analysis to perform well, examples being global value numbering and loop-invariant hoisting.

It is helpful for our purposes to describe the various meanings an alias relation can take, 'alias' can be defined as either must-alias or may-alias. In the former, aliasing occurs in all possible paths in the control flow graph, whereas in the latter that is only the case for some paths. As we will see in later sections, DDC employs the notion of distinctness to characterise aliasing among regions, which can be viewed as the complement of may-alias.

Traditional alias analysis for imperative languages also distinguishes between flow-sensitive and flow-insensitive aliasing, i.e. whether aliasing information changes between instructions. This is not relevant for our work since store properties cannot be changed in Disciple, thanks to the phase distinction between region variables and actual region handles [Lip09].

2.3.2 Alias Analysis

Literature on compilers and programming languages contains many approaches and algorithms to alias analysis. [DMM98] describes several type-based algorithms and presents promising results. Our work also relies on types to improve optimisation (notably we also focus on redundant load elimination for testing and evaluation), however we aim to encode aliasing information in the type system rather than use types to infer aliasing information.

2.4 The LLVM Compiler Framework

Originally developed by Chris Lattner [Lat02], the LLVM (formerly Low Level Virtual Machine) is now an open-source and powerful compiler framework with major industry support. The main goal of the project was to provide a multi-stage optimisation strategy that is modular and extensible; thus LLVM constitutes a good platform for code optimisation and generation.

Most importantly to us, LLVM can already perform a multitude of compile-time optimisations, many of which depend on the effectiveness of alias analysis. One of our goals is to aid those optimisations by presenting Disciple's knowledge of aliasing information to LLVM. This became possible with the introduction of type-based alias analysis metadata in LLVM 2.9.

2.4.1 The `metadata` Type

First we introduce general extensible metadata, as it appears in the version of LLVM we will be using, LLVM 3.1.

The motivation behind metadata is to add additional information for debugging and optimisation. Among LLVM types, `metadata` is not first-class, which means values of this type can only be arguments to intrinsics, operands to other metadata, at top-level in a module, or attached to an instruction [Lat10]. The last use case is particularly important, since it is how we will encode the connection between a region and its witnesses in LLVM.

Values of type `metadata` can be a:

- String (`MDString`): sequence of characters surrounded by quotes, e.g. `!"hello"`
- Node (`MDNode`): *numbered* tuple of some values. The syntax for declaring an `MDNode` is: `!`*number* `= !{`$operand_1, \ldots, operand_n$`}`, which can then be referred to with `!`*number*, e.g.

```
!1 = \{i32 42, !"hello"\}              -- MDNode of an int and an MDString
!2 = \{!"world", i64 0, metadata !1\}  -- MDNode of an MDString, an int and
                                          another MDNode
```

- Named Node (`NamedMDNode`): module-level named list of `MDNodes`, such that it is possible for clients of metadata to retrieve only metadata of interest to them, e.g. for the type-based alias analysis pass to access all `!tbaa` metadata nodes in a module.

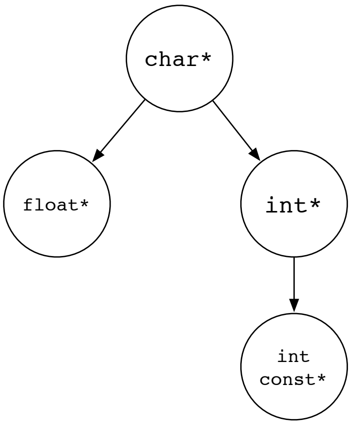
2.4.2 `!tbaa` Metadata

In LLVM IR, memory locations do not have types, thus `!tbaa` metadata was added to make type-based alias analysis possible. The design of `!tbaa` was motivated by the use case of representing the type hierarchy in C/C++, however with some coercion it can easily work with DDC's region discipline.

A `!tbaa` metadata node can have up to three fields [Lat12]:

- The first field contains a name that uniquely identifies the node within its own sub-tree (consequently two trees with different root node names are disjoint, regardless of the node names of their children).
- The second field indicates the parent of the node. A type is considered to alias:
  - All of its ancestors and descendants.
  - All types in all other trees.
- The third field, if present and is the integer `1`, indicates that the type is of a constant value.

The collection of all `!tbaa` nodes in an LLVM module forms a forest (set of trees). In C/C++, they can be used to represent the type hierarchy, e.g.

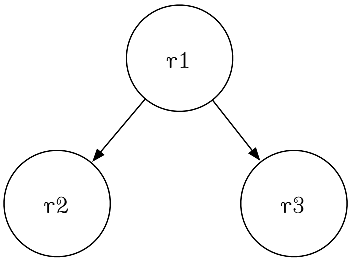


```
!0 = metadata !{metadata !"char*",  null, i32 0}
!1 = metadata !{metadata !"float*", metadata !0, i32 0}
!2 = metadata !{metadata !"int*",   metadata !0, i32 0}
!3 = metadata !{metadata !"int const*", metadata !0, i32 1}
```

Metadata node declarations in the module

Figure 2.2: `!tbaa` metadata representing the relationship among some primitive pointer types in C++ (i.e. `char*` can point to anything, `float*` cannot point to an *Int* value, etc.)

For Disciple, metadata trees can be used to represent regions and the relationship among them. For example, suppose we have a set of three regions $\mathcal{R} = \{r_1, r_2, r_3\}$ with the constraint *Const* r_3, where r_2 and r_3 are non-aliasing memory stores (Chapter 3 discusses how we define this relation). $\mathcal{R}$ and its aliasing data can be encoded with the following `!tbaa` metadata set:



```
!0 = metadata !{metadata !"r1", null, i32 0}
!1 = metadata !{metadata !"r2", metadata !0, i32 0}
!2 = metadata !{metadata !"r3", metadata !0, i32 1}
```

Metadata node declarations in the module

Figure 2.3: `!tbaa` metadata representing some regions in Disciple

The information provided by `!tbaa` metadata is used by the type-based alias analysis pass (TBAA) in LLVM to gather aliasing data, useful to several optimisation passes such as global value numbering.

3 Control Aliasing with Witnesses

3.1 Motivation

3.1.1 Optimisation Opportunities

Knowing aliasing properties of data leads to many possible optimisations. For example, consider a contrived function that may be used to update a value `v` given a counter `c`, returning the successor of `c`:

```
next = Λ r1 r2. λ (w1 : Mutable r1). λ (v : Int r1) (c : Int r2).
       letregion r3 in
       let (x : Int r3) = addInt    [r1] [r2] [r3] v c
           _            = updateInt [r1] [r2] <w1> v x
       in  addInt [r2] [r3] [r2] c (1 [r3] ())
```

Where the types of *addInt* and *updateInt* are:

$$\texttt{addInt}\ \ \ \ \texttt{:: [r1 r2 r3 : \%]. Int r1} \to \texttt{Int r2} \xrightarrow{\mathit{Read}\ r1\ +\ \mathit{Read}\ r2\ +\ \mathit{Alloc}\ r3\ \mid\ \mathit{Use}\ r1} \texttt{Int r3}$$

$$\texttt{updateInt :: [r1 r2 : \%]. Mutable r1} \Rightarrow \texttt{Int r1} \to \texttt{Int r2} \xrightarrow{\mathit{Read}\ r_2\ +\ \mathit{Write}\ r_1\ \mid\ \mathit{Use}\ r_1} \texttt{Unit}$$

Now suppose we apply local unboxing and let-floating on *next* with the following primitives:

$$\texttt{boxInt}\ \ \ \texttt{:: [r : \%]. Int\#} \xrightarrow{\mathit{Alloc}\ r\ \mid\ \mathit{Use}\ r} \texttt{Int r}$$

$$\texttt{unboxInt :: [r : \%]. Int r} \xrightarrow{\mathit{Read}\ r\ \mid\ \$0} \texttt{Int\#}$$

$$\texttt{add\#}\ \ \ \ \ \texttt{:: Int\#} \to \texttt{Int\#} \to \texttt{Int\#}$$

For clarity, each binding is annotated with its effect:

```
1  next = Λ r1 r2. λ (w1 : Mutable r1). λ (v : Int r1) (c : Int r2).
2         letregion r3 in
3         let (v1 : Int#)   = unboxInt  [r1] v                   Read r1
4             (c1 : Int#)   = unboxInt  [r2] c                   Read r2
5             (x  : Int r3) = boxInt    [r3] (add# v1 c1)        ⊥
6             _             = updateInt [r1] [r3] <w1> v x       Read r3 + Write r1
7             (c2 : Int#)   = unboxInt  [r2] c                   Read r2
8         in  boxInt r2 (add# c2 1#)                             ⊥
```

Since the variables r_1 and r_2 might refer to same region handle, the effects *Read* r_2 on line 4 could interfere with the *Write* r_1 on line 6, thus we cannot eliminate the duplicate unboxing on line 7. This effect interference also means that we cannot arbitrarily change the order of the bindings, unlike in Haskell. If, however, r_1 and r_2 are guaranteed to not alias, we can perform all of these optimisations.

A similar situation arises in the LLVM IR code generated by DDC's backend. LLVM has the capacity to remove redundant loads to memory, but can only do so if the store locations do not alias:

```
%a = load i32 %x
store %y %a
%b = load i32 %x
```

LLVM can easily remove the second load to `x` provided it has evidence that `x` and `y` do not alias.

3.1.2 Using *Const* Witnesses

One way to make sure *next* cannot ever be called with aliasing regions is to require r_2 to be a constant region. Since we have a witness that r_1 is a mutable region, we know that r_1 cannot alias r_2. This means *next* cannot be called with the same region, as Disciple's type system ensures that incompatible witnesses such as *Const r* and *Mutable r* cannot be created.

```
next = Λ r1 r2.λ (w1 : Mutable r1) (w2 : Const r2). λ (v : Int r1) (c : Int r2). ...

letregion r with { w1 : Mutable r; w2 : Const r } in       Not possible!
let (v : Int r1) = 42 [r] ()
    (c : Int r1) = 1  [r] ()
in  next [r] [r] <w1> <w2> v c
```

However this solution is stricter than necessary, as there is no reason why r_2 has to be constant other than to imply distinctness.

3.2 Witness of Distinctness

3.2.1 Construction

We instead define a new witness type that explicitly expresses distinctness. Distinctness can be thought of as a relation between two regions:

```
Distinct :: % ⇝ % ⇝ @
```

Where @ is the kind of witness types. In Core IR, we can construct a witness of distinctness as follows:

```
letregion r1 with { w1 : Const r1 } in
letregion r2 with { w2 : Distinct r1 r2 } in
letregion r3 with { w3 : Distinct r1 r3, w4 : Distinct r2 r3 } in
...
```

Returning to the previous example, we specialise *next* to require a *Distinct* witness, thus making it possible to eliminate the redundant unboxing:

```
next = Λ r1 r2.λ (w1 : Mutable r1) (w2 : Distinct r1 r2). λ (v : Int r1) (c : Int r2).
       letregion r3 with { w : Const r3 } in
       let (v1 : Int#)   = unboxInt [r1] v                    Read r1
           (c1 : Int#)   = unboxInt [r2] c                    Read r2
           (x  : Int r3) = boxInt [r3] (add# v1 c1)           ⊥
           _             = updateInt [r1] [r3] <w1> v x       Read r3 + Write r1
       in  boxInt r2 (add# c1 1#)                             ⊥
```

3.2.2 Typing Rules

Like *Const* and *Mutable* witnesses, the introduction of *Distinct* witnesses is restricted to the **letregion** construct, which means that to ensure the validity of distinct witnesses in the program, we only need to check for the well-formedness of the witness bindings in **letregion**. Recall that in Chapter 2, we list the constraints Disciple places on **letregion**:

- Uniqueness: If a **letregion** binds a variable r, only r can be used in the witness bindings.
- Well-formedness of region witnesses: The binders $\overline{w_i : \delta_i}$ cannot contain witnesses of types *Const* and *Mutable* for the same w_i.

For distinct witnesses, we change the uniqueness constraint to allow a witness binding to refer to other regions in scope, as long as one of the two arguments to the *Distinct* type constructor is the region being bound. This is the only condition needed since there exists no witness type in the system that can contradict a distinct witness.

We present the typing rules for **letregion** and witness bindings as follows. Note that the rule for **letregion** belongs in a larger set of rules, which defines typing of terms in the current Core IR, which is substantially different to those introduced in [Lip09] and [Lip10]. Although the typing rules for Core are incomplete and cannot be presented here, the reader can assume the basic rules are similar to the ambient System-F style λ-calculus upon which Disciple Core is based.

Symbols		
r	$\to$	(type variables)
Kinds		
κ	$\to$	(kinds)
$*$	$\to$	(data)
$\%$	$\to$	(regions)
Types		
τ	$\to$	(types)
σ	$\to$	(effects)
Read, Write, Alloc	$\to$	(effect type constructors)
$-$	$\to$	(effect type functions)
γ	$\to$	(closures)
cutT	$\to$	(closure type functions)
Environment		
$\Gamma,\ x : \tau$	$\to$	(type environment)
$\Delta,\ x : \kappa$	$\to$	(kind environment)

Figure 3.1: Fragment of Core syntax necessary to define **letregion**

$$\boxed{\Delta \,|\, \Gamma \;\vdash\; t : \tau \;;\; \sigma \;;\; \gamma}$$

$$\ldots$$

$$\frac{r \notin \Delta \qquad \Delta \,|\, r \vdash \overline{w_j : \tau_j} \text{ well-formed} \qquad \Delta,\ r : \% \,|\, \Gamma,\ \overline{w_j : \tau_j} \;\vdash\; t : \tau \;;\; \sigma \;;\; \gamma \qquad \Delta \;\vdash\; \tau : * \qquad r \notin fv(t)}{\Delta \,|\, \Gamma \;\vdash\; \textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_j : \tau_j}\}\ \textbf{in}\ t : \tau \;;\; \sigma - \mathit{Read}\ r - \mathit{Write}\ r - \mathit{Alloc}\ r \;;\; \mathit{cutT}\ r\ \gamma}$$

$$\textsc{TyLetR}$$

$$\ldots$$

Figure 3.2: Typing of **letregion** expression

This rule says that the expression **letregion** r **with** $\{\overline{w_j : \tau_j}\}$ **in** t is well-typed if and only if:

- The body t is well-typed, and its type τ is well-kinded.
- The set of witness bindings $\overline{w_j : \tau_j}$ is well-formed.
- The region being bound (r) does not shadow an existing one.
- r does not appear free in the body of t.

We also must take care to prevent the bound region r from escaping via the type of the expression by removing r from the effect and closure types.

Well-formedness of witness bindings in **letregion** is given by:

$$\boxed{\Delta \mid x \vdash \Gamma \text{ well-formed}}$$

$$\frac{}{\Delta \mid r \vdash \emptyset \text{ well-formed}}\ \textsc{WfEmpty}$$

$$\frac{\Delta \mid r \vdash \Gamma \text{ well-formed} \qquad \textit{Const}\ r \notin \Gamma}{\Delta \mid r \vdash \Gamma,\ w : \textit{Mutable}\ r \text{ well-formed}}\ \textsc{WfMutable}$$

$$\frac{\Delta \mid r \vdash \Gamma \text{ well-formed} \qquad \textit{Mutable}\ r \notin \Gamma}{\Delta \mid r \vdash \Gamma,\ w : \textit{Const}\ r \text{ well-formed}}\ \textsc{WfConst}$$

$$\frac{r_2 : \% \in \Delta \qquad r_1 \notin \Delta}{\Delta \mid r_1 \vdash \Gamma,\ w : \textit{Distinct}\ r_1\ r_2 \text{ well-formed}}\ \textsc{WfDistinct1}$$

$$\frac{r_2 : \% \in \Delta \qquad r_1 \notin \Delta}{\Delta \mid r_1 \vdash \Gamma,\ w : \textit{Distinct}\ r_2\ r_1 \text{ well-formed}}\ \textsc{WfDistinct2}$$

Figure 3.3: Well-formedness of witnesses

3.3 Extensions

Aside from the *Distinct* witness type itself, we also provide some additional Core language extensions which naturally arise from the use of distinct witnesses. There are a number of other useful extensions that we have planned but not yet implemented. These possible extensions are described in detail in Chapter 6.

3.3.1 Elaborate Implicit Properties

The evaluation of **letregion** replaces the region variable being bound with a fresh region handle. This guarantees that the new region does not alias any other region in scope, thus creating a *Distinct* witness only serves to bind a proof term (the witness) so that it can be given to function calls that require it, e.g.

```
Λ r1. λ (x : Int r1)
letregion r2 with { w1 : Distinct r1 r2 } in
letregion r3 with { w2 : Mutable r3 } in
let (y : Int r2) = 42 [r2] ()
in  f [r1] [r2] <w1> x y
...
f :: [r1 r2 : %]. Distinct r1 r2 → Int r1 → Int r2 → Unit
```

In the above example, it is clear that r_3 is distinct from both r_1 and r_2, however this information is not used explicitly in the program, so the witness was not created (by the user). For the sake of optimisation and accurate aliasing information for the LLVM back-end, we provide a transformation pass over the Core AST that inserts these 'implicit' distinct witnesses into the program. This is done by binding the proof term anonymously:

```
Λ r1. λ (x : Int r1)
letregion r2 with { w1 : Distinct r1 r2 } in
letregion r3 with { w2 : Mutable r3; _ : Distinct r1 r2, _ : Distinct r2 r3} in
...
```

Another implicit witness in this example is *Const* r_2. Since *Const* and *Mutable* are mutually exclusive properties of the store, we can assume that any non-mutable region is constant. Our transform can therefore insert a *Const* witness for any region that does not have a *Mutable* one:

```
Λ r1. λ (x : Int r1)
letregion r2 with { w1 : Distinct r1 r2; _ : Const r2 } in
letregion r3 with { w2 : Mutable r3; _ : Distinct r1 r2, _ : Distinct r2 r3 } in
...
```

This simple pass is implemented as one of the many transformations available in DDC's simplifier.

3.3.2 letregions

When writing Disciple Core IR, we have found that introducing multiple region variables is a popular idiom, which becomes even more prominent with the use of distinct witnesses:

```
letregion r1 in
letregion r2 with { w : Distinct r1 r2 } in ...
```

Allowing multiple regions to be constructed at the same time makes the expression much less verbose:

```
letregions r1 r2 with { w : Distinct r1 r2 } in ...
```

This new **letregions** construct has the same constraints that guarantee the well-formedness of the witnesses it introduces as the old **letregion**.

3.3.3 *Distinct*$_n$

We generalise the notion of distinct witnesses to include multi-way distinctness, i.e. a single proof term of type *Distinct*$_n$ that represents pair-wise distinctness over a set of n regions. In Core IR, we can now use the family of constructors:

```
Distinct3 :: % ⇝ % ⇝ % ⇝ @
Distinct4 :: % ⇝ % ⇝ % ⇝ % ⇝ @
...
```

For example, we can construct a three-way witness of distinctness as follows:

```
letregions r1 r2 r3 with { w : Distinct3 r1 r2 r3 } in ...
```

The well-formedness condition for *Distinct*$_n$ is more complicated than *Distinct* in the previous section. We need to ensure that all the regions involved, minus the one being introduced, are already mutually distinct. This requires searching the environment for evidence of distinctness for every single pair of regions. For example, in the following construction, to introduce a proof of four-way distinctness for $\{r_1,\ r_2,\ r_3, r_4\}$, we need to search the environment for either $\mathit{Distinct}_3\ r_1\ r_2\ r_3$ or $(\mathit{Distinct}\ r_1\ r_2) \wedge (\mathit{Distinct}\ r_2\ r_3)$.

```
Λ r1 r2 r3.
letregion r3 with { w : Distinct4 r1 r2 r3 r4 } in ...
```

This approach is not very practical due to the combinatorial explosion of the search space, so for the current implementation we restrict $\mathit{Distinct}_n$ to only be allowed in the **letregions** introducing all the arguments needed. Under this restriction, this example would not type check, but the previous one would check successfully.

4 Metadata Generation

In Chapter 2, we saw how LLVM represents the *alias* relation in metadata with trees (in the data structure sense, i.e. directed and rooted), where each node is considered to alias its ascendants and descendants, and all trees alias each other. We now discuss the problem of translating Disciple's notion of distinct witnesses to this tree-based relation.

To simplify the problem, we assume that all distinct witnesses in the input have type *Distinct* (distinctness between two regions). We can make this assumption without loss of generality since it is always possible to convert all $Distinct_n$ witnesses to a set of *Distinct* witnesses.

We choose to focus entirely on the 'alias' relation, as it can be trivially derived from Disciple's set of distinct witnesses: for any two regions r_1 and r_2, if there does not exist a witness for distinctness between them, then they are considered to alias each other.

Before devising a conversion of 'alias' between DDC and LLVM, we establish the goals of the operation. Let $\mathcal{S}$ be the set of regions, $\mathcal{A}$ be the alias relation in DDC and $\mathcal{A}'$ be the alias relation generated as a result of the conversion. Beside from being recognised by LLVM, the best construction must be:

- **Safe**: if two regions alias each other in the input, the conversion ensures they must alias in the output (otherwise there would potentially be unsafe optimisations done in LLVM), that is:

$$\forall\ r_1, r_2 \in \mathcal{S}\ (r_1 \neq r_2).\ (r_1, r_2) \in \mathcal{A} \implies (r_1, r_2) \in \mathcal{A}'$$

- **Optimal**: Of all 'alias' relations that satisfy the safety condition, $\mathcal{A}'$ is the smallest.

For reasons discussed in later sections, our conversion is *safe but not optimal.*

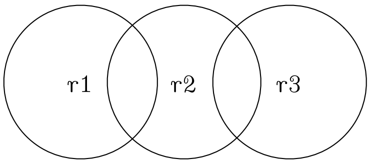
4.1 Solution

In DDC, two regions may alias each other if there does not exist a witness of distinctness for them. The properties of this binary relation are:

- Reflexivity - a witness of type *Distinct r r* cannot be constructed for any region r. However LLVM already assumes a memory location must alias itself, so we discard this information for the sake of simplicity.

- Symmetry - if r_1 alias r_2, then the intersection of two regions is not null, thus r_2 also aliases r_1.

- Intransitivity - consider the counterexample with regions r_1, r_2, r_3, where r_1 aliases r_2 and r_2 aliases r_3, but they do not imply r_1 alias r_3:



These properties allows us to represent 'alias' as the adjacency relation in the undirected graph where each node represents a region. As a concrete example, consider the set of regions $\mathcal{S} = \{r_1, r_2, r_3, r_4\}$

and distinct witnesses $\mathcal{D} = \{\textit{Distinct}\ r_1\ r_2,\ \textit{Distinct}\ r_3\ r_4\}$. The corresponding alias relation can be obtained by including any pair of regions that is not covered by a distinct witness:

$$\mathcal{A} = \{(r_1, r_3), (r_3, r_1), (r_1, r_4), (r_4, r_1), (r_2, r_3), (r_3, r_2), (r_2, r_4), (r_4, r_2)\}$$

$\mathcal{A}$ is represented as the adjacency relation in the following graph:

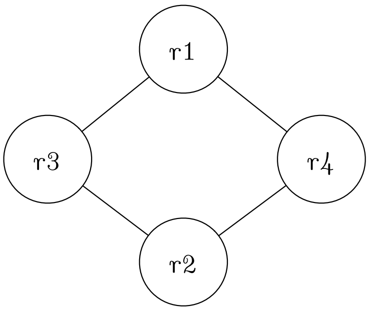


Figure 4.1: Undirected graph where adjacency is 'alias'

So far we have reduced our alias translation problem to the problem of safely and optimally converting an undirected graph to metadata trees, which can be further divided into two sub-problems:

(1) **Orientation**: assign a direction to the undirected graph.

(2) **Partition**: split the oriented graph into disjoint trees.

Here we make the assumption that, for the conversion to be optimal, the intermediate graph produced by orientation must also be optimal. In section 4.1.3, we explain the reason and justification for doing this.

4.1.1 Orientation Method

We define a notion of 'alias' for the intermediate directed graph, so that safety is verifiable. To do this, first we observe that metadata tree is a kind of directed graph, thus its descendant relation is equivalent to 'reachability', furthermore its ascendant relation is the same as reachability in the inverted version of itself. Since the reachability relation of a directed graph is the transitive closure of its arc set, we use the term "transitive closure" to mean the graph formed by the reachability relation of the original graph:

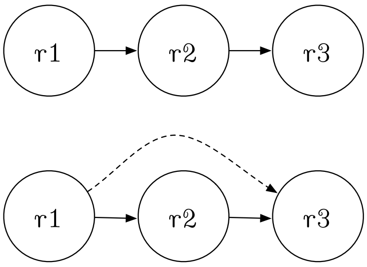


Figure 4.2: A directed graph and its transitive closure

Hence it is reasonable for us to define 'alias' in the intermediate directed graph as the union of the graph's transitive closure and its inversion (which refers to the transitive closure of the same graph but with all arcs inverted). This definition allows us to rephrase our two conditions in terms of transitive closures:

- Safety - The original alias relation must be contained within the union of the transitive closure of the resulting graph and its inversion. This is already true for all orientations, since the undirected graph of the original 'alias' can be seen as a directed graph with edges going both ways.
- Optimality - Let the size of a directed graph be the size of its arc set, the result must have the smallest transitive closure compared to all other orientations. In the best case, this is the orientation whose transitive closure is itself.

Before giving an example, we summarise the problem as: *given an undirected graph $\mathcal{G}$, find an orientation $\mathcal{D}$ for $\mathcal{G}$ such that the number of edges in the transitive closure of $\mathcal{D}$ is minimised.* Returning to the example graph $\mathcal{U}$ from before, we conceive some possible orientations:

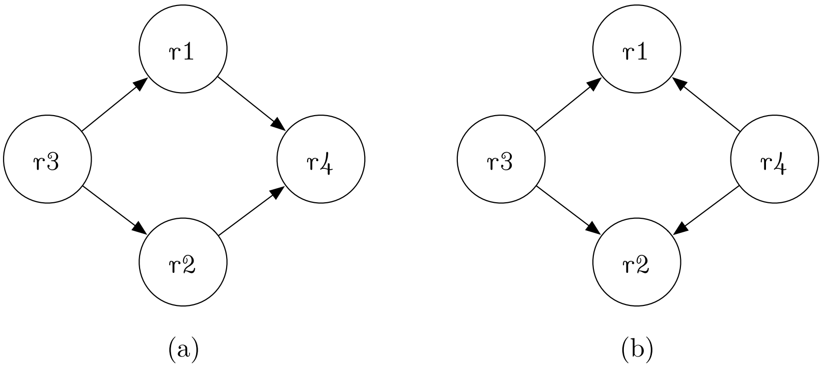


Figure 4.3: Possible orientations for the example undirected graph $\mathcal{U}$

The transitive closures of (a) and (b) are:

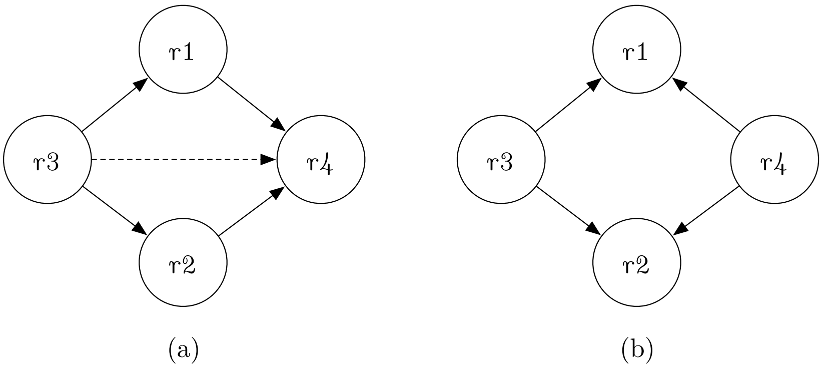


Figure 4.4: Transitive closures of the above orientations

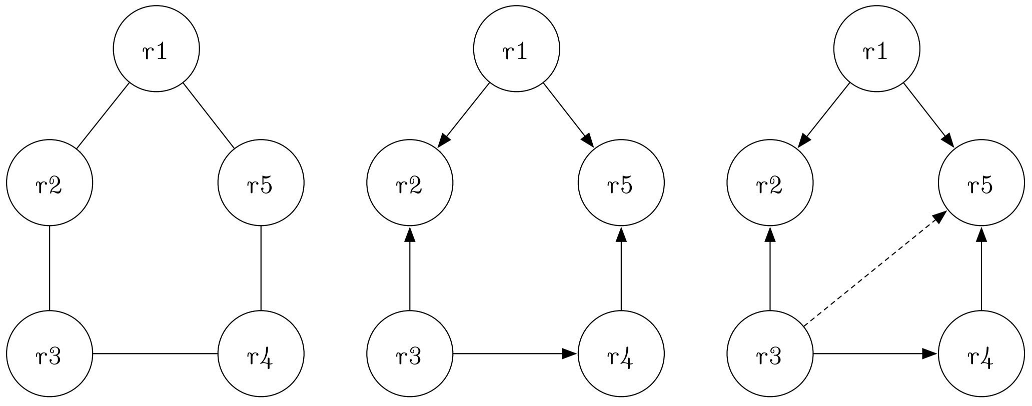
The transitive closure of (b) is smaller than (a), thus (b) is a better choice to represent the original undirected graph than (a). Indeed, (b) encodes the same amount of aliasing information as the original graph, while (a) represents a superset that includes the spurious edge (r_3, r_4).

Simply giving the undirected graph any orientation is sufficient for safety; but to satisfy optimality we must search for the orientation with the smallest closure possible. This requires sorting all orientations by the size of their transitive closure, which has a combinatorial effect in the size of the edge set. In our experiments, this brute-force method becomes infeasible for any edge set with a size larger than nine. As this is unacceptable for practical purposes, we devise a compromise solution:

- In the special case that there exists a solution where the transitive closure of the result is itself, we can solve the problem in linear time with the algorithm described in [MS97].

- Otherwise (i.e. the linear algorithm failed), for sets of regions smaller than a certain size, we use brute-force to find the optimal solution. For larger sets we simply choose a random orientation. The exact threshold varies from machine to machine, so we make it an option in the compiler.

In the above example, (b) is the optimal solution and can be found in linear time. An example where this is not the case and we have to brute-force for the optimal solution is:



(a) Original undirected graph

(b) Optimal orientation found with brute-force

(c) And its transitive closure

Figure 4.5: A case where the input has no orientation whose transitive closure is itself

4.1.2 Partition Method

Our next problem is to partition the directed graph obtained from the previous phase into disjoint tree structures for LLVM's alias analysis.

Given a directed graph $\mathcal{D}$, partition $\mathcal{D}$ into some k components ($k \geq 1$) and assign a dummy root to each component such that each component forms a tree structure, and the partition satisfies our conversion goals:

- Safety - To preserve the aliasing relation, if two nodes u, v are adjacent in the underlying undirected graph, then either:
  - u and v must be partitioned into different trees, since LLVM considers all trees to alias all other trees.
  - u must be an ascendant or descendant of v in the same tree, since LLVM considers a node to alias all of its ascendants and descendants.
- Optimality - The partition must be the one with the least aliasing induced among the nodes. The amount of aliasing in a partition is characterised by 'outer-aliasing', i.e. aliasing among all nodes in different trees, as well as 'inner-aliasing', i.e. aliasing among nodes in the same tree.
  - For any tree, the amount of 'outer-aliasing' is simply the product of the number of nodes in the tree and number of nodes not in the tree.
  - The amount of 'inner-aliasing' is the size of the descendant relation (or ascendant), which is equivalent to the size of the transitive closure of the tree.

  In the following example, the first tree's outer-aliasing is $|tree_1| * (|tree_2| + |tree_3|) = 12$ and inner-aliasing is 3 (the dummy roots are ignored):

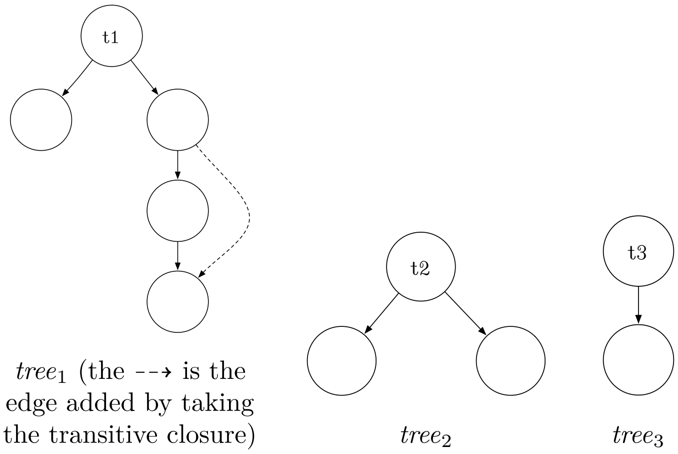
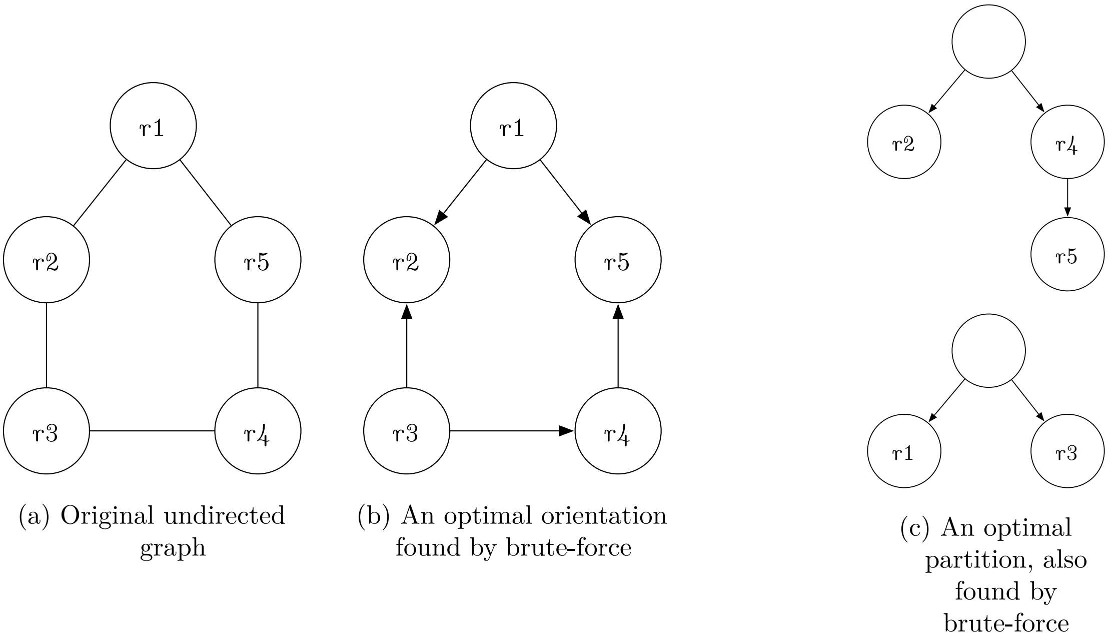

t1
t2
t3
tree$_1$ (the --→ is the edge added by taking the transitive closure)
tree$_2$
tree$_3$

Figure 4.6: Measure of aliasing in metadata trees

To compute the overall aliasing represent by a list of trees, we simply fold over the list to accumulate outer-aliasing, which is then added to the sum of all inner-aliasing values. In the above example, the overall aliasing amount is: (outer-aliasing) + (inner-aliasing) $= (|tree_1| * (|tree_2| + |tree_3|) + |tree_2| * |tree_3|) + (3 + 0 + 0) = 17$.

The easiest way to find such a partition is by exhaustively generate all possible partitions of $\mathcal{D}$, sort by the induced aliasing amount and choose the best one. Clearly this is not the best idea because of the combinatorial explosion of the search space (there are $\sum_{k=1}^{n} \binom{n}{k}$ possible partitions for a vertex set of size n). In our experiments this brute-force approach becomes infeasible for sets of regions larger than nine, so for practical purposes of the compiler we choose a random partition when the size of the edge set exceeds a certain threshold; a similar approach to our method for finding the orientation. Finding a better way of approximating the optimal partition is not critical, as we know that LLVM developers have plans to replace the tree representation for `!tbaa` metadata with directed acyclic graphs.

As a sanity-check example, consider the case in Figure 4.6 and the partition found by the brute-force method we described:



(a) Original undirected graph

(b) An optimal orientation found by brute-force

(c) An optimal partition, also found by brute-force

Figure 4.7: An full example of converting an undirected graph into a set of disjoint trees.

The original directed graph was derived from the set of distinct witnesses:

$$\{\mathit{Distinct}\ r_1\ r_3,\ \mathit{Distinct}\ r_1\ r_4,\ \mathit{Distinct}\ r_2\ r_4,\ \mathit{Distinct}\ r_2\ r_5,\ \mathit{Distinct}\ r_3\ r_5\}$$

The following pairs of nodes are distinct in the end-result:

$$\{(r_1, r_3), (r_2, r_4), (r_2, r_5)\}$$

Clearly the conversion did not introduce any new distinct pair, and thus is at least safe. Some information was lost in the translation (e.g. *Distinct* $r_1\ r_4$), but the current choice of trees for `tbaa` metadata in LLVM makes this inevitable.

4.1.3 Assumption

In the conversion described above, we made the assumption that optimality of the conversion depends on optimality of the orientation phase. We now discuss the consequences and justification for this decision.

Recall that our original problem was to convert the 'alias' relation in DDC to metadata trees in LLVM, such that the conversion is safe and optimal. To satisfy optimality, the most simple approach would involve considering all partitions of all orientations. The complexity of this brute-force method involves the product of two combinatorial factors (orientation and partition), hence it is not suitable for practical use. If, however, we make the assumption that optimality of the orientation operation is required for optimality of the overall conversion, then we only need to consider all partitions of the optimal orientation, which reduces the product to the sum. The problem with making this assumption is that the conversion might produce sub-optimal results. Consider the set of regions $\mathcal{S} = \{r_1, r_2, r_3, r_4, r_5\}$ and the alias relation:

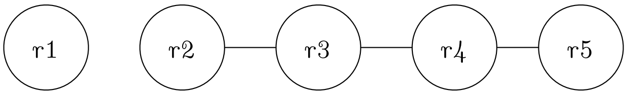


Figure 4.8: A graph that represents the alias relation $\{(r_2, r_3), (r_3, r_2), (r_3, r_4), (r_4, r_3), (r_4, r_5), (r_5, r_4)\}$

An optimal orientation is one that minimises the amount of aliasing induced by the edges:

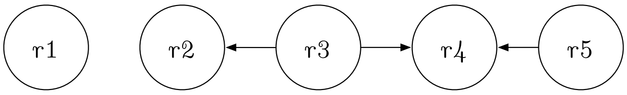


Figure 4.9: An optimal orientation for the above relation.

Likewise, an optimal partition minimises the amount of aliasing represented:

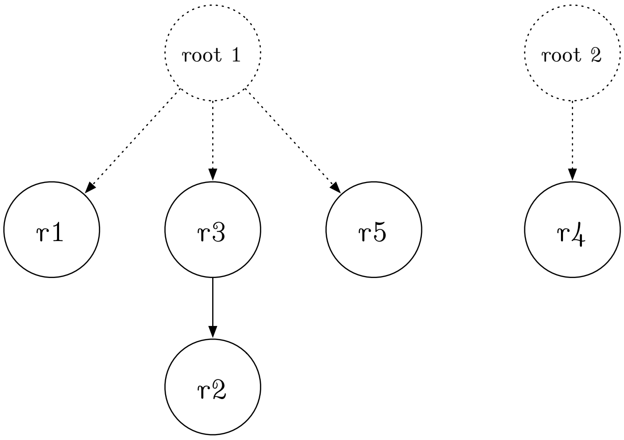


Figure 4.10: An optimal partition of the above orientation.

Here, the amount of outer-aliasing (aliasing among different trees) is $|tree_1| * |tree_2| = 4$ and inner-aliasing (aliasing in the same tree) is 1 (ignoring the root nodes). Hence overall this partition has an aliasing measure of 5.

It is however not the best representation of the original undirected graph — as shown by the counter-example:

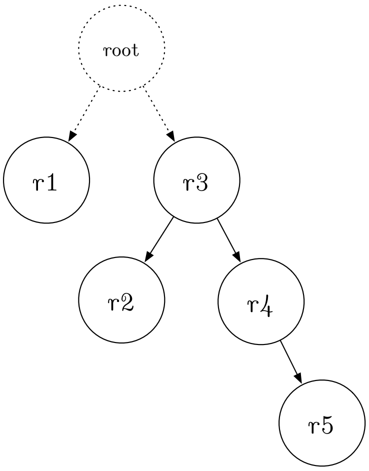


Figure 4.11: A better solution

This partition has only one tree, so its outer-aliasing is nil. Ignoring root nodes, the tree has an inner-aliasing of 4 (since there are four edges in the transitive closure), hence the overall aliasing measure of the partition is 4, which makes it a better partition than the previous one.

So, making this assumption gives better complexity at the cost of a sub-optimal result. This reason alone would be insufficient grounds to make the assumption, but another compelling motivation exists for doing so. According to our informal correspondence with LLVM developers, the tree implementation of `!tbaa` metadata is due to be changed in favour of a directed acyclic graph (DAG) representation. While this change is intended to support struct types in C, it is also good news for our purposes, since it effectively renders the partitioning half of our conversion problem obsolete. Once this change is implemented, finding the optimal orientation will be the only necessary step in our conversion. For this reason we choose to keep this assumption when devising the conversion.

4.2 Implementation and Testing

4.2.1 Implementation

We perform metadata conversion and annotation on a function level. In DDC's LLVM back-end, each generated LLVM function is associated with a list of metadata declarations to be pretty-printed with the function. For the purpose of annotating LLVM instructions with relevant metadata nodes, we also keep a mapping from region variables to metadata declarations.

Using an internal explicit representation for metadata trees, we separate the conversion of metadata from the generation of their declarations. Once a tree structure has been constructed from the region and witness bindings in the function, we simply traverse the tree breadth-first to generate metadata.

During the translation of Core Salt expressions to LLVM instructions, for any `load/store` generated, we search in the regions ↔ metadata mapping and attach any metadata concerning the region involved to the `load/store` instruction.

4.2.2 Correctness

We have provided intuitive justification for the safety of our conversion method. Short of an actual proof, we use QuickCheck, a specification-based testing library [CH00] to further establish the safety property for our implementation. The property is stated as "there exists no pair of regions that alias each other in DDC and not in the generated LLVM metadata":

```
prop_alias_safety (UG (d, aliasDDC))
  = null [ (x,y) | x <- d, y <- d, aliasDDC x y, not (aliasLLVM x y) ]
```

Where `aliasLLVM` is true for any two regions `x` and `y` if either `x` and `y` are in different trees or they are ascendant/descendant of each other in the same tree.

Since optimality is not guaranteed for our conversion, we do not check for it.

5 Optimisation Results

In this chapter we present some preliminary results that showcase the effect of using Disciple's aliasing information to aid optimisation in LLVM. We use some small examples to study at a low level how some transform passes use the additional aliasing information supplied by our metadata.

5.1 Global Value Numbering

Global Value Numbering (GVN) is an optimisation on the SSA form that assigns symbolic keys to computations such that computations with the same key compute the same value [RWZ88].

5.1.1 Load Elimination

LLVM's GVN pass (-gvn) relies on alias analysis to eliminate redundant loads. For example, consider:

```
%1 = load i32 %x
store i32 %1, i32 %y
%2 = load i32 %x
```

If x and y can be determined to not alias, then the second load to x will be eliminated. We simulate this scenario with a tiny example in Core Salt IR (the final form of Disciple code before conversion to C or LLVM IR (see Chapter 2 for more details on the Salt language itself). We make use of the primitive integer type (Int#), the primitive pointer type (Ptr#), along with the peek# and poke# pointer operations:

```
peek# :: [r : %].[t : *].Ptr# r t → Nat# → t
poke# :: [r : %].[t : *].Ptr# r t → Nat# → t → Void#
```

peek# dereferences the argument pointer at the given offset, while poke# updates the pointer with a new value, e.g.

```
-- Reads a pointer to a memory object in region r1, with offset 3
peek# [r1] [Obj]  x 3#;

-- Updates a pointer to an Int in region r2 with the new value 5
poke# [r2] [Int#] y 5 0#;
```

Additionally, we use the primitive add# and mul# operations for arithmetic addition and multiplication respectively. The Core Salt language is significantly more cumbersome to use than Core IR, but at the level of optimisation detail we are looking at, it provides an easy way to write and tweak examples without the obstruction of the transforms done in Core.

Listing 5.1: GVN-alias: small example of the effect *Distinct* witnesses have on GVN

```
x_plus_y_square
    [rx ry rz : %]                          -- Takes three regions
    <w1 : Distinct rx rz>                   -- Two witnesses
    <w2 : Distinct ry rz>
    (x : Ptr# rx Int#)                      -- Three value arguments
    (y : Ptr# ry Int#)                      -- of type pointer to int
    (z : Ptr# rz Int#)
    : Int#
 = do { xval1 = peek# [rx] [Int#] x 0#;     -- Compute (x + y)
        yval1 = peek# [ry] [Int#] y 0#;
        a     = add# [Int#] xval1 yval1;

        poke# [rz] [Int#] z 0# a;           -- Modify z

        xval2 = peek# [rx] [Int#] x 0#;     -- Compute (x + y) again
        yval2 = peek# [ry] [Int#] y 0#;
        b     = add# [Int#] xval2 yval2;

        mul# [Int#] a b;                    -- Result is (x + y)^2
      };
```

This function simply computes the square of the sum of its first two arguments (the presence of the third argument is contrived, added for the sake of generating a non-boring metadata tree). Since `x`, `y` and `z` are all pointers, without a proof of distinctness `z` is assumed to alias `x` and `y`, and thus the update to `z` can affect the values of `x` and `y`. However, since we have witnesses of distinctness between `x`/`y` and `z`, we know for sure this is not the case, hence there is no need to recompute `x + y` after the modification of `z`. With this aliasing information encoded in metadata, GVN should be able to make the changes accordingly:

Listing 5.2: Generated LLVM IR for the small GVN-alias example

```
define external ccc i64 @x_plus_y_square(i64*  %x, i64*  %y, i64*  %z) align 8  {
l6.entry:
        (...Compute the pointer offset for peek# x...)
        %xval1       = load i64* %xval1.ptr,    !tbaa !3
        (...Compute the pointer offset for peek# y...)
        %yval1       = load i64* %yval1.ptr,    !tbaa !4
        %a           = add i64 %xval1, %yval1

        (...Compute the pointer offset for poke# z...)
        store i64 %a, i64* %_v9.ptr,    !tbaa !5                    -- modify z

        (...)
        %xval2       = load i64* %xval2.ptr,    !tbaa !3            -- redundant load
        (...)
        %yval2       = load i64* %yval2.ptr,    !tbaa !4            -- redundant load
        %b           = add i64 %xval2, %yval2

        %_v10        = mul i64 %a, %b
        ret i64 %_v10
}

!5 = metadata !{metadata !"x_plus_y_square_rz", metadata !2, i32 0}
!4 = metadata !{metadata !"x_plus_y_square_ry", metadata !3, i32 0}
!3 = metadata !{metadata !"x_plus_y_square_rx", metadata !2, i32 0}
!2 = metadata !{metadata !"x_plus_y_square_ROOT_1", null, i32 1}
```

The LLVM IR generated by our backend is quite verbose; here we elided bits of it that are irrelevant to the topic at hand. The metadata tree produced is:

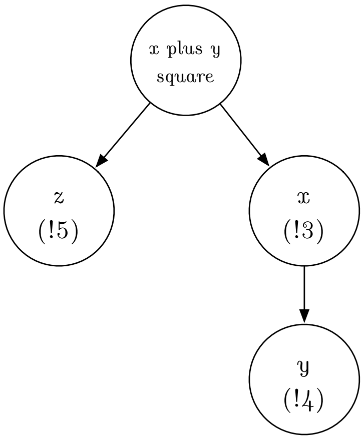


Figure 5.1: Metadata tree for GVN aliasing example

We ran LLVM's optimiser on the generated code and observe the differences in the output. We tell LLVM to collect aliasing information from metadata and to perform GVN with: `opt gvn-alias.ll -S -tbaa -basicaa -gvn`. The result is:

Listing 5.3: Optimised LLVM IR for GVN-alias example

```
define i64 @x_plus_y_square(i64* %x, i64* %y, i64* %z) align 8 {
l6.entry:
        (...Compute the pointer offset for peek# x...)
        %xval1 = load i64* %xval1.ptr, !tbaa !0
        (...)
        %yval1 = load i64* %yval1.ptr, !tbaa !2
        %a = add i64 %xval1, %yval1

        (...Compute the pointer offset for poke# z...)
        store i64 %a, i64* %_v9.ptr, !tbaa !3                        -- modify z

        %_v10 = mul i64 %a, %a
        ret i64 %_v10
}

!0 = metadata !{metadata !"x_plus_y_square_rx", metadata !1, i32 0}
!1 = metadata !{metadata !"x_plus_y_square_ROOT_1", null, i32 1}
!2 = metadata !{metadata !"x_plus_y_square_ry", metadata !0, i32 0}
!3 = metadata !{metadata !"x_plus_y_square_rz", metadata !1, i32 0}
```

The two `loads` after the update to `z` were eliminated as expected, consequently the computation of `b` was discarded as well. By contrast, if the metadata is removed, then GVN cannnot perform the optimisation, as reported in its debug messages:

```
GVN: load i64 %xval2 is clobbered by   store i64 %a, i64* %_v9.ptr
GVN: load i64 %yval2 is clobbered by   store i64 %a, i64* %_v9.ptr
```

This result proves the potential of using *Distinct* witnesses to aid GVN. One drawback in this approach is that, since only `load` and `store` instructions can be annotated, the computation of `x + y` has to be done inside the main body, i.e. we cannot use a function to add the two numbers without inlining it:

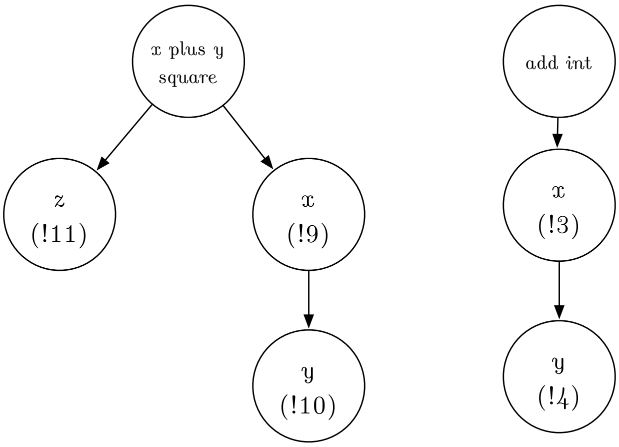
Listing 5.4: Non-inlined version of GVN-alias example

```
add_int
    [rx ry : %]
    (x : Ptr# rx Int#) (y : Ptr# ry Int#)
    : Int#
 = do { xval1 = peek# [rx] [Int#] x 0#;
        yval1 = peek# [ry] [Int#] y 0#;
        add# [Int#] xval1 yval1;
      };

x_plus_y_square
    (...Elided type...)
 = do { a = add_int [rx] [ry] x y;
        poke# [rz] [Int#] z 0# a;
        b = add_int [rx] [ry] x y;
        mul# [Int#] a b;
      };
```

The above code compiles to:

```
(...Omitted code for add_int...)

define external ccc i64 @x_plus_y_square(i64*  %x, i64*  %y, i64*  %z) align 8  {
l12.entry:
        %a           = call i64 @add_int (i64* %x, i64* %y)
        (...)
        %_v14.addr2  = add i64 %_v13.addr1, 0
        (...)
        store i64 %a, i64* %_v15.ptr,    !tbaa !11
        %b           = call i64 @add_int (i64* %x, i64* %y)
        (...)
}
```

And the metadata:



The `add_int` function calls cannot be annotated with our `tbaa` metadata, and even when they are inlined by LLVM, there is no way for LLVM to tell that it should replace the attached metadata nodes from `add_int`'s tree with nodes from `x_plus_y_square`'s. Hence if we keep metadata generation function-level, we need to rely on DDC's optimiser to perform inlining and let-floating transforms in order to expose these optimisations to LLVM.

5.1.2 Constant Propagation

GVN can also eliminate duplicate loads of constant memory. Consider another small example:

Listing 5.5: GVN-const: example of the effect *Const* witnesses have on GVN

```
  nothing [rx : %] (x : Ptr# rx Int#) : Int# = 42i#;

  three_x_plus_one
      [rx ry : %]
      <w : Const rx>
      (x : Ptr# rx Int#)
      : Int#
    = do { a = peek# [rx] [Int#] x 0#;
           b = add# [Int#] a 1i#;
           nothing [rx] x;               -- Unknown function, potentially unsafe
           c = peek# [rx] [Int#] x 0#;
           d = mul# [Int#] c 2i#;
           add# [Int#] b d;
         };
```

Since w provides evidence that x points to constant memory, the second peek# should be eliminated. The above code compiles to:

Listing 5.6: Generated LLVM IR for GVN-const example

```
1  define external ccc i64 @nothing(i64*  %x) align 8  { ... }
2
3  define external ccc i64 @three_x_plus_one(i64*  %x) align 8  {
4  l9.entry:
5          (...)
6          %a            = load i64* %a.ptr,    !tbaa !8
7          %b            = add i64 %a, 1
8          %_v10.dummy  = call i64 @nothing (i64* %x)
9          (...)
10         %c            = load i64* %c.ptr,    !tbaa !8
11         (...)
12 }
13
14 !8 = metadata !{metadata !"three_x_plus_one_rx", metadata !7, i32 1}
15 !7 = metadata !{metadata !"three_x_plus_one_ROOT_6", null, i32 1}
```

GVN does indeed remove the load on line 10; on the other hand if there was no metadata to guarantee the constancy of x, then it fails to do so:

```
GVN: load i64 %c is clobbered by   %_v10.dummy = call i64 @nothing(i64* %x)
```

5.2 Loop Invariant Code Motion

Loop Invariant Code Motion (LICM), or loop hoisting, is an important optimisation technique that detects loop-independent computations and moves them out of the loop [Muc97].

LLVM's LICM pass uses alias analysis for two purposes [SH10]:

- Moving loop invariants, i.e. determining if a load or call inside a loop does not alias anything that is ever modified, and hoisting them out of the loop. As with GVN, the problem with not being able to attach a metadata node to a `call` instruction prevents us from testing out LICM's ability to move calls, so we instead focus on loads.

- Scalar promotion of memory: If either the destination of a `store` is loop invariant, or if there are no `load` or `store` instructions in the loop which affect memory that may alias the destination, and there exist no calls that could modify or reference this destination, then the `store` instructions inside the loop can be moved to a post-loop block.

To test the effect of *Distinct* witnesses on these two optimisations, we create a simple example in DDC Core Salt IR:

Listing 5.7: LICM-alias: Example of the effect *Distinct* witnesses have on LICM

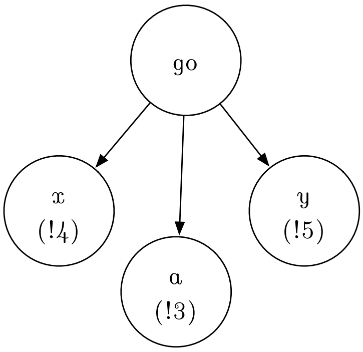
```
go  [ra rx ry : %]                              -- Takes three regions
    <w : Distinct3 ra rx ry>                    -- that are pair-wise distinct
    (a : Ptr# ra Nat#)                          -- Three pointers to natural numbers
    (x : Ptr# rx Nat#)
    (y : Ptr# ry Nat#)
    (i : Nat#)                                  -- Counter
    : Nat#
= case i of {
    42# -> i;
    _   ->
      do { yval     = peek# [ry] [Nat#] y 0#;   -- Loop invariant computation
           yplustwo = add# [Nat#] yval 2#;

           poke# [rx] [Nat#] x 0# yplustwo;     -- Loop invariant computation

           poke# [ra] [Nat#] a i i;             -- Loop-dependant computation

           nexti    = add# [Nat#] i 1#;
           go [ra] [rx] [ry] <w> a x y nexti;   -- Tail recursion
         };
  };
```

This is a simple recursive function that can be easily tail-call optimised with LLVM's `tailcallelim`. We placed two loop-independent computations in the recursive case, one involves a read-pointer operation that can be hoisted out of the loop (moving invariant loads), the other is a pointer modification that can be sunk of the loop (scalar promotion). The LLVM IR produced is shown in the following figure.

Listing 5.8: Generated LLVM IR for LICM-alias example

```
1  define external ccc i64 @go(i64*  %a, i64*  %x, i64*  %y, i64  %i) align 8  {
2  l6.entry:
3          switch i64 %i, label %l9.default [ i64 42,label %l7.alt ]
4  l7.alt:
5          ret i64 %i
6  l9.default:
7          (...)
8          %yval        = load i64* %yval.ptr,    !tbaa !5
9          %yplustwo    = add i64 %yval, 2
10         (...)
11         store i64 %yplustwo, i64* %_v12.ptr,    !tbaa !4
12         (...)
13         store i64 %i, i64* %_v15.ptr,    !tbaa !3
14         %nexti       = add i64 %i, 1
15         %_v16        = call i64 @go (i64* %a, i64* %x, i64* %y, i64 %nexti)
16         ret i64 %_v16
17 }
18
19 !5 = metadata !{metadata !"go_ry", metadata !2, i32 0}
20 !4 = metadata !{metadata !"go_rx", metadata !2, i32 0}
21 !3 = metadata !{metadata !"go_ra", metadata !2, i32 0}
22 !2 = metadata !{metadata !"go_ROOT_1", null, i32 1}
```

With the boring metadata tree:



Line 8 and 9 are the computations involving `y` that we wish to see moved to a pre-loop block, while line 11 is the modification of `x` that should be sunk of the loop.

LICM depends on a few other passes to realise the optimisations it can perform. With some experimentation we discover these passes are: `simplifycfg` - control flow graph simplification, `early-cse` - common subexpression elimination, and `loop-rotate` - simple loop rotation. The final command we used is: `opt licm.ll -S -tailcallelim -tbaa -basicaa -simplifycfg -early-cse -loop-rotate -licm -debug`. With `debug`, we can ensure the transformations we wanted were performed by LICM. The optimised code is shown in the following figure.

Listing 5.9: Optimised LLVM IR LICM-alias example

```
define i64 @go(i64* %a, i64* %x, i64* %y, i64 %i) align 8 {
l6.entry:
  %cond1 = icmp eq i64 %i, 42
  br i1 %cond1, label %l7.alt, label %l9.default.lr.ph

  l9.default.lr.ph:                       ; preds = %l6.entry
  (...)
  %yval = load i64* %yval.ptr, !tbaa !0
  %yplustwo = add i64 %yval, 2
  (...)
  br label %l9.default

tailrecurse.l7.alt_crit_edge:           ; preds = %l9.default
  %split = phi i64 [ %nexti, %l9.default ]
  store i64 %yplustwo, i64* %_v12.ptr
  br label %l7.alt

l7.alt:                                 ; preds = %tailrecurse.l7.alt_crit_edge, %l6.entry
  %i.tr.lcssa = phi i64 [ %split, %tailrecurse.l7.alt_crit_edge ], [ %i, %l6.entry ]
  ret i64 %i.tr.lcssa

l9.default:                             ; preds = %l9.default.lr.ph, %l9.default
  %i.tr2 = phi i64 [ %i, %l9.default.lr.ph ], [ %nexti, %l9.default ]
  %_v14.addr2 = add i64 %_v13.addr1, %i.tr2
  %_v15.ptr = inttoptr i64 %_v14.addr2 to i64*
  store i64 %i.tr2, i64* %_v15.ptr, !tbaa !2
  %nexti = add i64 %i.tr2, 1
  %cond = icmp eq i64 %nexti, 42
  br i1 %cond, label %tailrecurse.l7.alt_crit_edge, label %l9.default
}

!0 = metadata !{metadata !"go_ry", metadata !1, i32 0}
!1 = metadata !{metadata !"go_ROOT_1", null, i32 1}
!2 = metadata !{metadata !"go_ra", metadata !1, i32 0}
```

As we expected, the computations involving `y` were moved out to the loop initialisation block (`default.lr.ph`) and the store to `x` was moved to the post-loop block (`tailrecurse`), while the loop-dependant store to `a` is kept within the loop body. By contrast, these changes were not made when we removed the metadata. The debug messages confirmed that it is indeed LICM that did the optimisations:

```
(...)
LICM hoisting to l9.default.lr.ph:   %yplustwo = add i64 %yval, 2
(...)
LICM: Promoting value stored to in loop:   %_v12.ptr = inttoptr i64 %_v10.addr1 to i64*
```

These experimental results proved the usefulness of type-level witnesses in DDC to code optimisation. So far we have only studied GVN and LICM, but there are other optimisation passes in LLVM that could benefit from witnesses such as `sccp` (sparse conditional constant propagation). Devising a more comprehensive benchmark suite is part of our future work.

6 Conclusions

6.1 Related Work

6.1.1 Region-Effect Systems

MLKit

The other significant work based on Talpin and Jouvelot's type and effect system is the MLKit compiler [BLE+06]. MLKit combines Standard ML with a region-based memory management model to track the allocation of memory at runtime. Disciple also uses regions for this purpose, but additionally it also tracks mutability and aliasing properties of regions, as part of an analysis to guide compile-time optimisations.

Some optimisations have been implemented for MLKit and presented in [BTV96]. Test programs compiled with MLKit ran between ten times faster and four times slower than with the standard SML compiler at the time (SML of New Jersey); and it became apparent that for their system to produce practical speed-ups, the test programs need to be modified to be "region-friendly", a process that requires understanding of the MLKit analyses themselves [TBEH04].

By contrast, we are not targeting memory allocation based optimisations, rather we aim to utilise the mutability and aliasing information encoded in the type system to aid existing optimisations.

Call-by-value λ-calculus

One of the first uses of region types was in Tofte and Talpin's implementation of the call-by-value λ-calculus [TT94]. The main goal of this work is to improve space characteristics of functional programs, and help alleviate the overhead that comes with complete dependence on garbage collection. Significant performance improvements to this system were made in [AFL95].

Calculus of Capabilities

A later related project, the calculus of capabilities (CC), is a compiler intermediate language with support for region-based memory management [CWM99]. The contribution of CC is the use of static capabilities to specify permissions of operations such as memory access and deallocation. Cyclone is a type-safe dialect of C developed based on CC. Capabilities to an object in CC can be revoked and CC can statically ensure that revoked capabilities are not used. This is used in Cyclone's region system to manage deallocation of memory. In DDC, a capability (e.g. the ability to read from a region) cannot be revoked by the user, and regions are not used to track the lifetime of an object as in Cyclone.

In contrast to both of these works, the main use of regions in DDC is to track mutability and aliasing properties, not as a memory management mechanism.

6.1.2 LLVM Front-ends

GHC LLVM Backend

Previous work on the LLVM back-end for GHC (Glasgow Haskell Compiler) has produced robust and promising results; although there were no significant speed-ups achieved with LLVM compared to the C and NCG (Native Code Generator) back-ends [TC10]. The authors conjectured that this is because the Cmm output is not easily optimised (Cmm is a C-like intermediate representation used by GHC).

Further investigation has identified the unsatisfactory alias analysis done by LLVM for GHC as one of the main reasons for this average performance; currently there is continuing work on improving the situation in the Haskell community, most notably a custom alias analysis part dedicated to GHC. The addition of `!tbaa` metadata has been noted by the developers, however no clear results have been documented of which we are aware.

Clang

Aside from the experimental work in GHC, the only compiler of which we are aware that utilises LLVM's new `!tbaa` metadata is Clang, a compiler front-end for C/C++ and Objective-C/Objective-C++. Reportedly the new type-based alias analysis has allowed significant better load/store optimisations in Clang [Lat11].

6.2 Future Work

6.2.1 $Distinct_n$ Witness Operators

Split

Recall that we define $Distinct_n$ to be a generalisation of *Distinct*, consisting of a family of type constructors:

```
Distinct3 :: % ⇝ % ⇝ % ⇝ @
Distinct4 :: % ⇝ % ⇝ % ⇝ % ⇝ @
...
```

This means a witness of type $Distinct_3\ r_1\ r_2\ r_3$ cannot be used where a witness of type $Distinct\ r_1\ r_2$ is expected, even though it is clear that $Distinct_3\ r_1\ r_2\ r_3 \implies Distinct\ r_1\ r_2$. We plan to provide the Core language with a witness constructor that performs this conversion. The constructor, tentatively named **split**, is a function that takes a list of regions and a witness:

$$\textbf{split} :: \forall\ \mathcal{R} : [\%].\ Wit \to Wit$$

We give an informal notion of the type for **split**, as it is polymorphic over the arity of the $Distinct_n$ constructor and thus current non-trivial to type within Disciple's type system. Given the set of regions $\mathcal{R}$ and the witness w, if the type of w is $Distinct_n\ r_0 \ldots r_n$ and $\mathcal{R}$ is a subset of $\{r_0, \ldots, r_n\}$, then **split** will construct a distinct witness that represents mutual distinctness for all elements in $\mathcal{R}$. The type checker ensures that this is the case whenever **split** is used. An example use of **split** would be:

```
Λ r1 r2 r3.
λ (w : Distinct3 r1 r2 r3).
λ (x : Int r1) (y : Int r2).
f [r1] [r2] <split [[r1,r2]] w> x y
...

f :: [r1 r2 : %]. Distinct r1 r2 ⇒ Int r1 → Int r2 → Unit
```

Note that in `[[r1,r2]]`, the outer brackets represent a type application, while the inner brackets are intended to represent a set of regions. This implementation of **split** is not the most convenient or intuitive, a more user-friendly version of the operation should infer the witness type needed, so that the user only needs to supply the superset witness, e.g.

```
f [r1] [r2] <split w> x y
```

However this is not suitable for the Core language, which we intend to keep simple enough that a proof of soundness is feasible. Inference and constraint solving is done during the compilation from Source to Core, so a more user-friendly **subset** could be implemented as part of the Source language.

Combine

The natural dual of **split** is **combine**, which we define as witness constructor that takes a list of witnesses and produces a new one:

$$\textbf{combine} :: [\,\mathit{Wit}\,] \rightarrow \mathit{Wit}$$

Given the set of distinct witnesses $\mathcal{W}$, where each witness w_i can be thought to represent a total distinct relation on a set of regions $\mathcal{R}_i$ (a relation A is total in $\mathcal{S}$ if $\forall x, y \in \mathcal{S}, xAy \vee yAx$), **combine** will determine if there exists a total distinct relation on the set $\mathcal{R} = \bigcup_i \ \mathcal{R}_i$, in which case a witness will be produced to express it.

In Core IR, **combine** is used in a similar fashion to **split**:

```
Λ r1 r2 r3.
λ (w1 : Distinct r1 r2) (w2 : Distinct r2 r3) (w3 : Distinct r1 r3).
λ (x : Int r1) (y : Int r2) (z : Int r3).
f [r1] [r2] [r3] <combine [w1,w2,w3]> x y z
...

f :: [r1 r2 r3 : %]. Distinct3 r1 r2 r3 ⇒ Int r1 → Int r2 → Int r3 → Unit
```

6.2.2 Derive

Consider the scenario where a function call requires a witness of distinctness, but the caller only has witnesses for constancy and mutability:

```
Λ r1 r2.
λ (w1 : Const r1) (w2 : Mutable r2).
λ (x : Int r1) (y : Int r2).
f [r1] [r2] <?> x y
...

f :: [r1 r2 : %]. Distinct r1 r2 ⇒ Int r1 → Int r2 → Unit
```

Since Disciple's type system guarantees that witnesses of constancy and mutability cannot be created for the same region, we know from the existence of w_1 and w_2 that r_1 and r_2 do not alias, however we need a concrete witness of distinctness to convince `f` of this fact. This is possible with **derive** (for lack of a better name), a witness constructor that converts *Const* r_1 and *Mutable* r_2 to $\mathit{Distinct}_2\ r_1\ r_2$:

```
derive :: [r1 r2 : %]. Const r1 ⇒ Mutable r2 ⇒ Distinct r1 r2
```

We can then use **derive** to construct the distinct witness required:

```
f [r1] [r2] <derive w1 w2> x y
```

The three constructors, **split**, **combine** and **derive** can be used in tandem quite nicely:

```
Λ r1 r2 r3.
λ (w1 : Const r1) (w2 : Mutable r2) (w3 : Mutable r3) (w4 : Distinct r1 r3).
λ (x : Int r1) (y : Int r2) (z : Int r3).
f [r2] [r3] <split [[r2,r3]] (combine [w4, derive w1 w2, derive w1 w3])> y z
...

f :: [r1 r2 : %]. Distinct r1 r2 ⇒ Int r1 → Int r2 → Unit
```

Here, the application of *f* requires the witness *Distinct* r_2 w_3, which can be constructed as follows: **derive** $\langle w_1 \rangle$ $\langle w_2 \rangle$ creates a witness for *Distinct* r_1 r_2, and **derive** $\langle w_1 \rangle$ $\langle w_3 \rangle$ creates *Distinct* r_1 r_3, which can be used in combination with w_4 to conceive $\mathit{Distinct}_3\ r_1\ r_2\ r_3$. Then with **split**, we can extract *Distinct* r_2 r_3 as needed.

6.2.3 Integration with Rule-based Rewriting

[Rob12] describes a rule-based rewrite system for DDC that could benefit from our support for distinct witnesses. For example, we can have specialised versions of primitive (or standard library) functions that depend on distinct witnesses:

```
copyVector  :: [r1 r2 : %]. Vector r1 a → Buffer r2 a
copyVector' :: [r1 r2 : %]. Distinct r1 r2 → Vector r1 a → Buffer r2 a
```

The above functions both copy the contents of some vector to a fresh buffer. In the latter, r_1 and r_2 are guaranteed to not alias in any way, thus it is possible for the implementation to be very aggressive performance-wise, e.g. to perform the copy in parallel. This rewrite system can then easily replace uses of `copyVector` with its optimised counterpart whenever possible.

6.2.4 Better Metadata Conversion

In Chapter 4, we discussed the challenge of finding the optimal orientation for Disciple's 'alias' undirected graph. Recall that the solution in the best case is the orientation whose transitive closure is itself, also known as the *transitive orientation* of the original undirected graph. Finding the transitive orientation, if it exists, can be done in linear-time [MS97]. When the 'alias' graph has no transitive orientation, we have to fall back on either exhaustively search for the best possible orientation or choosing a random one. None of these options is a very good choice, fortunately we can rely on the transitive orientation once more to approximate the best solution.

The key idea is the equivalence between exhaustively searching for the orientation with the smallest transitive closure and finding the minimum number of edges to add to the 'alias' graph such that it is transitively orientable. The latter is known as the *minimum comparability completion* of the original undirected graph, where a *comparability completion* is defined as a graph obtained from adding edges to the original graph. We (informally) show the correspondence between the two by contradiction. For an ‘alias’ graph $\mathcal{A}$, suppose that there exists an orientation $\mathcal{O}_1$ with a smaller transitive closure than the transitive orientation of the minimum completion, $\mathcal{O}_2$. Let $\mathcal{O}_1^+$ be the transitive closure of $\mathcal{O}_1$, obviously the transitive closure of $\mathcal{O}_1^+$ is itself, thus $\mathcal{O}_1^+$ is the transitive orientation of its underlying undirected graph. This underlying undirected graph contains all the edges in $\mathcal{A}$, hence it is a comparability completion of $\mathcal{A}$. However, by assumption this graph has fewer edges than the minimum completion of $\mathcal{A}$, which makes it a smaller completion than the minimum completion, a contradiction. Therefore $\mathcal{O}_2$ is the orientation with the smallest transitive closure. We conclude that the two problems are equivalent.

The correspondence by itself is not very useful, as finding the minimum comparability completion of an undirected graph is NP-hard [HSY97]. We can nonetheless benefit from the research done on approximating the minimum completion. [HMP06] presents a polynomial time algorithm to find the *minimal comparability completion*, which they define as “a comparability completion $\mathcal{H}$ of $\mathcal{G}$ such that no proper subgraph of $\mathcal{H}$ is a comparability graph of $\mathcal{G}$”. Their algorithm provides a way to approximate the minimum completion efficiently, which we intend to implement as part of our metadata conversion.

Any future work on metadata conversion is of course subjected to changes in the way LLVM implements `!tbaa` metadata. If the proposed replacement of trees with DAGs is implemented in LLVM, our conversion will be able to preserve much more of Disciple’s aliasing information than it currently does.

6.2.5 Benchmarking

In this work we have studied at a very low-level some simple examples to observe the effect of using distinct witnesses to control aliasing. While this is sufficient for a first experiment, to accurately quantify the practical benefits of our method we need a more comprehensive benchmark suite with more practical and complex example programs. Ideally we should be able to automate checking of the number of loads eliminated, number of invariants hoisted, etc.

6.3 Summary of Contributions

- A new *Distinct* witness type for DDC that expresses the no-alias relation between two regions, together with new typing rules for the well-formedness of witnesses.
- Some extensions to the Core language, including changing the **letregion** construct to enable simultaneous introduction of multiple regions and a transform pass to create witnesses for axiomatic region properties.
- A conversion from *Distinct* witnesses to `!tbaa` metadata in LLVM via the LLVM back-end, with some regard to the future of `!tbaa` in LLVM.
- Preliminary results that showcase the potential of distinct witnesses in aiding LLVM transforms.

We have shown that type systems can be used to control aliasing with high precision, as opposed to the traditional approach of using static analysis to infer aliasing information. There is much work to be done to explore the full potential of region-effect typing with regard to optimisation, our result is a step towards this goal.

I do not pretend to start with precise questions. I do not think you can start with anything precise. You have to achieve such precision as you can, as you go along.

— Bertrand Russell, *The Philosophy of Logical Atomism* (1918)

Bibliography

[AFL95] Alexander Aiken, Manuel Fahndrich, and Raph Levien. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages (Extended Abstract). In *In ACM Conference on Programming Language Design and Implementation*, pages 174–185. ACM Press, 1995.

[BLE$^+$06] Mads Tofte Birkedal, Lars, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with Regions in the MLKit (Revised for Version 4.3.0). Technical report, January 2006.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183, New York, NY, USA, 1996. ACM.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed Memory Management in a Calculus of Capabilities . In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '99*, pages 262–275, New York, New York, USA, 1999. ACM Press.

[DMM98] Amer Diwan, Kathryn S McKinley, and J Eliot B Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 106–117, New York, NY, USA, 1998. ACM.

[HMP06] Pinar Heggernes, Federico Mancini, and Charis Papadopoulos. Making arbitrary graphs transitively orientable: minimal comparability completions. In *Proceedings of the 17th international conference on Algorithms and Computation*, pages 419–428, Berlin, Heidelberg, 2006. Springer-Verlag.

[HSY97] S Louis Hakimi, Edward F Schmeichel, and Neal E Young. Orienting graphs to optimize reachability. *Inf. Process. Lett.*, 63(5):229–235, 1997.

[Lat02] Chris Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization.* PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002.

[Lat10] Chris Lattner. Extensible Metadata in LLVM IR, April 2010.

[Lat11] Chris Lattner. LLVM 2.9 Release Notes, April 2011.

[Lat12] Chris Lattner. LLVM Language Reference, October 2012.

[Lip09] Ben Lippmeier. Witnessing Purity, Constancy and Mutability. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, pages 95–110, Berlin, Heidelberg, 2009. Springer-Verlag.

[Lip10] Ben Lippmeier. *Type Inference and Optimisation for an Impure World.* PhD thesis, Australian National University, 2010.

[Lip12] Ben Lippmeier. *Disciple Core Language*, February 2012.

[LW91] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–302, New York, NY, USA, 1991. ACM.

[MS97] Ross M McConnell and Jeremy P Spinrad. Linear-time transitive orientation. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 19–25, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[Muc97] Steven S Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Rob12] Amos Robinson. Rewrite rules for the Disciplined Disciple Compiler (DDC). Undergraduate honours thesis, University of New South Wales, 2012.

[RWZ88] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global Value Numbers and Redundant Computations. In *Proc. of the 15th Symposium on Principles of Programming Languages.* ACM, 1988.

[SH10] Reid Spencer and Gordon Henriksen. *LLVM's Analysis and Transform Passes (3.0)*, March 2010.

[TBEH04] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A Retrospective on Region-Based Memory Management. *Higher Order Symbol. Comput.*, 17(3):245–265, 2004.

[TC10] David A Terei and Manuel M T Chakravarty. An LLVM backend for GHC. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 109–120, New York, NY, USA, 2010. ACM.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. In *Proc. of Logic in Computer Science*, pages 162–173. IEEE, 1992.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*, pages 188–201, New York, New York, USA, 1994. ACM Press.

[WB89] Philip L Wadler and S Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.