Data-flow and alias analysis for functional programming languages

NICKY ASK LUND, Aalborg University, Denmark

As ReScript introduces a strongly typed language that targets JavaScript, as an alternative to gradually typed languages, such as TypeScript. While ReScript is built upon OCaml, it provides its own build system and integration with JavaScript, as such not much analysis has been introduced to ReScript. They do provide an experimental analysis tool to analyze areas, such as dead-code and termination.

As data-flow analysis has been used for decades in compiler optimization, as they provide information about the data-flow in programs. As many languages use locations, the data-flow analysis must consider aliasing to ensure safety.

In this paper, we present a type system for data-flow analysis for a subset of the ReScript language, more specific for a λ-calculus with mutability and pattern matching. The type system is a local analysis that collects information about what variables are used and alias information.

Additional Key Words and Phrases: Data-flow analysis, Alias analysis, Program analysis, Programming languages, Type systems

ACM Reference Format:
Nicky Ask Lund. 2023. Data-flow and alias analysis for functional programming languages. J. ACM 1, 1, Article 1 (January 2023), 25 pages.

Contents

Author's address: Nicky Ask Lund, nlund18@student.aau.dk, Aalborg University, Institute of Computer Science, Aalborg, Denmark.

0004-5411/2023/1-ART1
https://doi.org/

1 INTRODUCTION

Data-flow analysis has been studied for decades to better to provide flow information of programs. This flow information has been used for different tasks for compiler optimization, debugging and understanding programs, testing and maintenance. In the context of compiler optimization, where the flow information provides data that may be used at given parts of the program at runtime.

The classical way of doing data-flow analysis has been by using iterative algorithm based on representing the control-flow of programs as graphs. The purpose of such graphs is to give a sound over-approximation of the control flow of a program, where edges represent the flow and nodes represent basic blocks. By using the information of control-flow graphs, many algorithms have been developed to use those by annotating the graphs and solving the maximal fix-point[? ?]. (Ref to algorithms, such as kilders) Other techniques have also been presented, such as a graph-free approach [?] or through type systems with refinement types [?].

When analysing languages, such as C/C++ or other languages that explicitly handles pointers, it is important to take into account aliasing, i.e., multiple variables referring to the same location. Many data-flow analysis uses alias algorithms to compute this information. Two overall types of alias algorithm has been used, flow-sensitive which give precise information but are expensive, and flow-insensitive which are less precise but are inexpensive [? ?].

This paper will focus on data-flow analysis with focus on a subset of the functional language ReScript, a new language based on OCaml with a JavaScript inspired syntax which targets JavaScript. ReScript offers a robust type system based on OCaml, which provides an alternative to other gradually typed languages that targets JavaScript.[?]

As ReScript provides integration with JavaScript, it provides its own compiler toolchain and build system for optimizing and compile to JavaScript. ReScript does, however, introduce an analysis tool for dead-code, exception, and termination analysis, but the tool is only experimental.[?] As ReScript introduces mutability, through reference constructs for creation, reading, and writing,

We will the present a type system for data-flow analysis for bindings and alias analysis. As type system have been used to provide a semantic analysis of programs usually used to characterize specific type of run-time errors. Type systems are implemented as either static or dynamic analysis, i.e., on compile time or run-time. Type systems are widely used, from weakly typed languages such as JavaScript, to strongly typed languages commonly found in functional languages such as Haskell and Ocaml.

This work is a generalization of [?], which focused on dead-value analysis. We will present the analysis for a language based on a subset of the ReScript language, for a λ-calculus with mutability, local bindings, and pattern matching. The type system we proposes provides the data information used at each program point and the alias information used. Since the analysis we present focus collecting dependencies that are used to evaluate a part of a program, we present a local analysis of programs.

We will first present the language, its syntax and semantics, in section 2 and the type system for data-flow analysis in section 3. Then we will present the soundness of the type system in section 4, and lastly we will conclude in ??.

2 LANGUAGE

This section will introduce a functional programming language, based on a subset of ReScript. As this is a generalization of a dead-value analysis system, the language presented here is based on the one found in [?]. The language we present is basically a λ-calculus with bindings, pattern matching and mutability. As the purpose of the dependency analysis is to analyse each subexpression of a program and differentiate them, the language is extended with labelling, which we also call program points, all expressions and subexpressions. When labelling a syntactical element or semantic element, we call it an occurrence, such that the analysis is done for an occurrence and its sub-occurrences, while the semantic occurrences are variables and locations.

In the language we assume that all local bindings, and recursive bindings, are unique, which can be ensured by using α-conversion on an occurrence. We also make a distinction between labelled and unlabelled expressions, such that we call occurrences as labelled expression, and we call unlabelled expressions as expressions.

In this section we will first formally introduce the abstract syntax for the language, where we will then present binding models. Then we will present the dependency function, to model the semantic flow-data, and lastly we will present the semantic as a big-step operational semantics.

2.1 Syntax

This section introduces the abstract syntax of the language, based on the one presented in [?]. The syntactic categories for the language is defined as:

$p \in \mathrm{P}$	– The category for program points
$e \in \mathrm{Exp}$	– The category for expressions, or unlabelled occurrencens
$o \in \mathrm{Occ}$	– The category for occurrences, or labelled expressions
$c \in \mathrm{Con}$	– The category for constants
$x,\ f \in \mathrm{Var}$	– The category for variables
$\updownarrow \in \mathrm{Loc}$	– The category for constants

We also introduce a notation for occurrences of categories where, for a category cat, we write cat^{P} to denote the pair $cat \times \mathrm{P}$, for occurrences as such: $\mathrm{Exp}^{\mathrm{P}} = \mathrm{Exp} \times \mathrm{P}$.

Since the category for occurrences are labelled expressions, it can further be defined as:

$$\mathrm{Occ} = \mathrm{Exp}^{\mathrm{P}}$$

The formation rules is then presented in fig. 1.

$$
\begin{array}{lcl}
\textit{Occurrence } o & ::= & e^{p} \\
\\
\textit{expression } e & ::= & x \mid c \mid o_1\ o_2 \mid \lambda x.o \\
& \mid & c\ o_1\ o_2 \\
& \mid & \text{let } x\ o_1\ o_2 \\
& \mid & \text{let rec } x\ o_1\ o_2 \\
& \mid & \text{case } o_1\ \tilde{\pi}\ \tilde{o} \\
& \mid & \text{ref } o \mid o_1 := o_2 \mid\ !o \\
\\
\textit{Pattern } s & ::= & n \mid b \mid x \mid \_
\end{array}
\qquad
\begin{array}{lcl}
\textit{Constant } c & ::= & n \mid b \\
& \mid & \mathit{PLUS} \\
& \mid & \mathit{MINUS} \\
& \mid & \mathit{TIMES} \\
& \mid & \mathit{EQUAL} \\
& \mid & \mathit{LESS} \\
& \mid & \mathit{GREATER} \\
\\
\textit{Patterns } \tilde{\pi} & ::= & (s_1, \cdots, s_n) \\
\\
\textit{Occurrences } \tilde{o} & ::= & (e_1^{p_1}, \cdots, e_n^{p_n})
\end{array}
$$

Fig. 1. Abstract syntax

Some notable constructs is further explained below.

Abstractions $\lambda\, x.o$ denotes an abstraction, with a parameter x and body o.

Constants c are either natural numbers n, boolean values b, or functional constants. We introduce a function *apply*, that for each functional constant c returns the result of applying c to its arguments.

$$\mathit{apply}(\mathit{PLUS}, 2, 2) = 2 + 2$$

Bindings let $x\ o_1\ o_2$ and let rec $f\ o_1\ o_2$, also called local declarations, are immutable bindings that binds the variables x to values o_1 evaluates to. We also introduces non-recursive and recursive bindings, by using the rec keyword.

Reference ref o is the construct for creating references which are handled as locations and allows for binding locations to local declarations. We also introduces constructs for reading from references, $!o$, and writing to references, $o_1\ :=\ o_2$.

Pattern matching case $o_1\ \tilde{\pi}\ \tilde{o}$, matches an occurrence with the ordered set, $\tilde{\pi}$, of patterns. For each pattern in $\tilde{\pi}$ there is also an occurrence in $\tilde{o}$, as such, both sets must be of equal size. We also denote the size of patterns as $|\tilde{\pi}|$ and the size of occurrences as $|\tilde{o}|$.

Example 2.1. Consider the following occurrence:

$$(\text{let x } (\text{ref } 3^{1})^{2}\ (\text{let y } (\text{let z } (5^{3})^{4}\ (\text{x}^{5} := \text{z}^{7})^{8})^{9}\ (!\text{x})^{10})^{11})^{12}$$

Here, we first creates a reference to the constant 3 and binds this reference to x (Such that x is an alias of this reference). Secondly we create a binding for y, where create a binding z, to the constant 5, before writing to the reference, that x is bound to, to the value that z is bound to. Lastly, we read the reference that x is bound to, where we expect to retrieve the value 5.

Next we defined the notion of free variables, in the usual way for λ-calculus, as follows:

Definition 2.1 (Free variables). The set of free variables is a function $fv : \text{Occ} \to \mathbb{P}(\text{Var})$, given inductively by:

$$\begin{aligned}
fv(x^p) &= \{x\} \\
fv(c^p) &= \emptyset \\
fv([\lambda\ y.e^{p'}]^p) &= fv(e^{p'})\backslash\{y\} \\
fv([e_1^{p_1}\ e_2^{p_2}]^p) &= fv(e_1^{p_1}) \cup fv(e_2^{p_2}) \\
fv([c\ e_1^{p_1}\ e_2^{p_2}]^p) &= fv(e_1^{p_1}) \cup fv(e_2^{p_2}) \\
fv([\text{let } y\ e_1^{p_1}\ e_2^{p_2}]^p) &= fv(e_1^{p_1}) \cup fv(e_2^{p_2})\backslash\{y\} \\
fv([\text{let rec } f\ e_1^{p_1}\ e_2^{p_2}]^p) &= fv(e_1^{p_1}) \cup fv(e_2^{p_2})\backslash\{f\} \\
fv([\text{case } e^{p'}\ (s_1, \cdots, s_n)\ (e_1^{p_1}, \cdots, e_n^{p_n})]^p) &= fv(e^{p'}) \cup fv(e_1^{p_1}) \cup \cdots \cup fv(e_n^{p_n})\backslash(\tau(s_1) \cup \cdots \cup \tau(s_n)) \\
fv([\text{ref } e^{p'}]^p) &= fv(e^{p'}) \\
fv([!e^{p'}]^p) &= fv(e^{p'}) \\
fv([e_1^{p_1}\ :=\ e_2^{p_2}]^p) &= fv(e_1^{p_1}) \cup fv(e_2^{p_2})
\end{aligned}$$

where $\tau(s)$, for a pattern s, is denoted as:

$$\tau(s) = \begin{cases} \{x\} & \text{if } s = x \\ \emptyset & \text{otherwise} \end{cases}$$

2.2 Environments and stores

We will now introduce the binding model used in the semantics, where we will present the environments and stores. Since the language we focus on introduces mutability, through the referencing, this needs to be reflected in our bindings model. Here, the referencing constructs can also be seen as how locations, or pointers, are created and handled, as such we introduce notion of stores to describe how they are bound.

Since this language is a λ-calculus, the environment keeps the bindings we currently know and as such the environment is a function from variables to values. The set of values, Values, is comprised by:

- All constants are values.
- Locations are values.
- Closures, $\langle x, e^{p'}, env\rangle$ are values.
- Recursive closures, $\langle x, f, e^{p''}, env\rangle$, are values.
- Unit values, (), are values.

Where a value $v \in \text{Values}$ is an expression given by the following formation rules:

$$v ::= c \mid \updownarrow \mid \langle x, e^{p'}, env\rangle \mid \langle x, f, e^{p''}, env\rangle \mid ()$$

Definition 2.2. The set of all environments, Env, is the set of partial functions from variables to values, given as:

$$\text{Env} = \text{Var} \rightharpoonup \text{Values}$$

Where $env \in \text{Env}$ denotes an arbitrary environment in Env.

Definition 2.3 (Update of environments). Let $env \in \text{Env}$ be an environment. We write $env[x \mapsto v]$ to denote the environment env' where:

$$env'(y) = \begin{cases} env(y) & \text{if } y \neq x \\ v & \text{if } y = x \end{cases}$$

We also introduce a function which, for a given value v, returns all variables that is bound to v.

Definition 2.4 (inverse env). Let v be a value and $env \in \text{Env}$ be an environment, the inverse function env^{-1} is then given as:

$$env^{-1}(v) = \{x \in dom(env) \mid env(x) = v\}$$

The store is a function that keeps the location bindings currently known. We also introduce a placeholder *next*, that represents the next free location.

Definition 2.5. The set of all stores, Sto, is the set of partial functions from locations, and the *next* pointer, to values, given as:

$$\text{Sto} = \text{Loc} \cup \{next\} \rightharpoonup \text{Values}$$

Where $sto \in \text{Sto}$ denotes an arbitrary store in Sto.

Definition 2.6 (Update of stores). Let $sto \in \text{Sto}$ be a store. We write $sto[\updownarrow \mapsto v]$ to denote the store sto' where:

$$sto'(\updownarrow_1) = \begin{cases} env(\updownarrow_1) & \text{if } \updownarrow_1 \neq \updownarrow \\ v & \text{if } \updownarrow_1 = \updownarrow \end{cases}$$

We also assume the existence of a function $new : \text{Loc} \to \text{Loc}$, which takes a location and finds the next location. This function is used on the location *next* points to, to get a new free location, which is not already bound in our store.

2.3 Dependencies

The goal of the collection semantics is to collect the semantic dependencies as they appear in a computation. To this end, we use a dependency function that will tell us for each variable and location occurrence what other, previous occurrences they depend upon.

As such, we use the dependency function to model the semantic flow of dependencies in an occurrence, where we present and ordering between those occurrences to denote the flow.

Definition 2.7 (Dependency function). The set of dependency functions, W, is a set of partial functions from location and variable occurrences to a pair of dependencies, such that:

$$\text{W} = \text{Loc}^P \cup \text{Var}^P \rightharpoonup \mathbb{P}(\text{Loc}^P) \times \mathbb{P}(\text{Var}^P)$$

A lookup in a dependency function w is for an element $u^p \in \text{Loc}^P \cup \text{Var}^P$, such that:

$$w(u^p) = (\{\updownarrow_1^{p_1}, \cdots, \updownarrow_n^{p_n}\}, \{x_1^{p'_1}, \cdots, x_m^{p'_m}\})$$

This should be read as: a lookup of an occurrence u^p, a variable or location occurrence, returns a pair of location and variable occurrences. We also denote the pair, retrieved from the dependency function, which we call a dependency pair such that (L, V) contains a set of location occurrences $L = \{\updownarrow_1^{p_1}, \cdots, \updownarrow_n^{p_n}\}$ and a set of variable occurrences $V = \{x_1^{p'_1}, \cdots, x_m^{p'_m}\}$.

Definition 2.8 (Update of dependency functions). Let $w \in \mathrm{W}$ be a dependency function and u^p be either a variable or location occurrence. We write $w[u^p \mapsto (L, V)]$ to denote the dependency function w' where:

$$w'(v^q) = \begin{cases} w(v^q) & \text{if } v^q \neq u^p \\ (L, V) & \text{if } v^q = u^p \end{cases}$$

Example 2.2. Consider the occurrence from example 2.1, where we can infer the following bindings for a dependency function w_{ex} over this occurrence:

$$w_{ex} = [x^2 \mapsto (\emptyset, \emptyset), z^4 \mapsto (\emptyset, \emptyset), y^9 \mapsto (\emptyset, \{x^5\}), \updownarrow^2 \mapsto (\emptyset, \emptyset), \updownarrow^8 \mapsto (\emptyset, \{z^7\})]$$

Where $\updownarrow$ is the location created from the reference construct. Here, we can see that the variable bindings are distinct, an the location $\updownarrow$ is bound multiple times, for the program points 2 and 8.

If we want to read a variable or location in w_{ex}, we must also know for which program point since there can exists multiple bindings for the same variable or location.

By considering example 2.2, we would like to read the information from the location, that x is an alias to. As it is visible from the occurrence in example 2.1, we know that we should read from $\updownarrow^8$, since we wrote that reference at the program point 8. We can also see that from w_{ex} alone it is not possible to know which occurrence to read, since there are no order defined between the bindings. We then present the notion of ordering, as a binary relation over program points:

Definition 2.9. Let P be a set of program points in an occurrence. Then $\sqsubseteq$ is a binary relation of P, such that:

$$\sqsubseteq \subseteq \mathrm{P} \times \mathrm{P}$$

Since we are interested in the ordering of the elements in a dependency function w, we will define an instantiation of definition 2.9. Since w, is a function from occurrences to a pair of occurrences, we first present a function for getting the program points from a set of occurrences:

Definition 2.10 (Occurring program points). Let O be a set of occurrences, then $\mathit{points}(O)$ is given by:

$$\mathit{points}(O) = \{p \in \mathrm{P} \mid \exists e^p \in O\}$$

With definition 2.10 defined, we present the instantiation of definition 2.9 over a dependency function w:

Definition 2.11. Let $w \in \mathrm{W}$ be a dependency function. Then $\sqsubseteq_w$ is given by:

$$\sqsubseteq_w \subseteq \{(p, p') \mid p, p' \in (\mathit{points}(\mathit{dom}(w) \cup \mathit{points}(\mathit{ran}(w))))\}$$

As the dependency function w is a model of which occurrences an occurrence is dependent on, the relation on w should also model the order a value is evaluated in, as such we define the partial order over a dependency function.

Definition 2.12 (Partial order of w). Let $w \in \mathrm{W}$ be a dependency function and $\sqsubseteq_w$ be a binary relation over w. We say that w is partial order if $\sqsubseteq_w$ is a partial order.

Example 2.3. Consider the example from example 2.2, if we introduce a binary relation over the dependency function w_{ex}, such that:

$$\sqsubseteq_{w_{ex}} = \{(2, 4), (2, 9), (5, 9), (2, 8), (8, 2)\}$$

From this ordering, it is easy to see the ordering of the elements. The ordering we present also respects the flow the occurrence from example 2.1 would evaluate to. We then know that the dependencies for the reference (that x is an alias to) is for the largest binding of $\updownarrow$.

As presented in definition 2.7 and definition 2.11, the dependency function and the binary relation are used to define the flow of information. As illustrated by example 2.3, we need to lookup the greatest of $\sqsubseteq_w$.

We first present a generic function for the greatest binding of a relation $\sqsubseteq$ of program points.

Definition 2.13 (Greatest binding). Let u be an element, either a variable or location, and S be a set of occurrences, then $uf(u, S)$ is given by:

$$uf(u, S) = \inf\{u^p \in S \mid u^q \in S.q \sqsubseteq p\}$$

Based on definition 2.13, we can present an instantiation of the function for the dependency function w and an order over w, $\sqsubseteq_w$:

Definition 2.14. Let w be a dependency function, $\sqsubseteq_w$ be an order over w, u be an element, that is either a variable or location, then $uf_{\sqsubseteq_w}$ is given by:

$$uf_{\sqsubseteq_w}(u, w) = \inf\{u^p \in dom(w) \mid u^q \in dom(w.q \sqsubseteq_w p\}$$

Example 2.4. As a continuation of example 2.3, we can now lookup the greatest element for an element, e.g., a variable or location. As we were interested in finding the greatest bindings a location is bound to in $w_e x$, we can now use the function $uf_{\sqsubseteq_w}$:

$$uf_{\sqsubseteq_{w_e x}}(\updownarrow, w_e x) = \inf\{\updownarrow^p \in dom(w) \mid \updownarrow^q \in dom(w).q \sqsubseteq_{w_e x} p\}$$

Where the set we get for $\updownarrow$ are as follows: $\{\updownarrow^2, \updownarrow^8\}$. From this, we find the greatest element:

$$\updownarrow^7 = \inf\{\updownarrow^2, \updownarrow^8\}$$

As we can see, from the $uf_{w_e x}$ function, we got $\updownarrow^8$ which were the occurrence we wanted.

2.4 Collection semantics

We will now introduce the big-step semantics for our language and highlight some interesting transition rules. In the big-step semantics, the transitions are of the from:

$$env \vdash \left\langle e^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle v, sto', (w', \sqsubseteq'_w), (L, V), p'' \right\rangle$$

Where $env \in$ Env, $sto \in$ Sto, and $w \in$ W. This should be read as: given the store sto, a dependency function w, A relation over w, and the previous program point p, the occurrence $e^{p'}$ evaluates to a value v, an updated store sto', an updated dependency function w', a relation over w', the dependency pair (L, V), and the program point p'' reached after evaluating $e^{p'}$, given the bindings in the environment env.

The transition system is given by:

$$((\text{Occ} \cup \text{Values}) \times \text{Store} \times (\text{W} \times (\text{P} \times \text{P}) \times \text{P}, \rightarrow, \text{Values} \times \text{Store} \times (\text{W} \times (\text{P} \times \text{P})) \times \mathbb{P}(\text{Loc}^P \times \text{Var}^P) \times \text{P})$$

A highlight of the rules for $\rightarrow$ can be found in fig. 2, the rest can be found in ??.

(Const) rule, for the occurrence $c^{p'}$, is the simplest rule, as it has no premises and does not have any side effects. As constants are evaluated to the constant value, no dependencies are used, i.e., no variable or location occurrences are used to evaluate a constant.

(Var) rule, for the occurrence $x^{p'}$, uses the environment to get the value x is bound to and uses dependency function w to get its dependencies. To lookup the dependencies, the function $uf_{\sqsubseteq_w}$ is used to get the greatest binding a variable is bound to, in respect to the ordering $\sqsubseteq_w$. Since the occurrence of x is used, it is added to the set of variable occurrences we got from the lookup of the dependencies for x.

(Let) rule, for the occurrence $[\text{let } x\ e_1^{p_1}\ e_2^{p_2}]^{p'}$, creates a local binding that can be used in $e_2^{p_2}$. The (Let) rule evaluate $e_1^{p_1}$, to get the value v, that x will be bound to in the environment for $e_2^{p_2}$, and the dependencies used to evaluate $e_1^{p_1}$ is bound in the dependency function. As we reach the program point p_1 after evaluating $e_1^{p_1}$, and it is also the program point before evaluating $e_2^{p_2}$, the binding of x in w is to the program points p_1.

(Ref) rule, for the occurrence $[\text{ref } e^{p'}]^{p''}$, creates a new location and binds it in the store sto, to the value evaluated from $e^{p'}$. The (Ref) rule also binds the dependencies, from evaluating the body $e^{p'}$, in the dependency function w at the program point p''. As the (Ref) rule creates a location (where we get the location from the $next$ pointer), and binds it in sto. The environment is not updated as (Ref) does not in itself give any alias information. To create an alias for a location, it should be bound to a variable using the (Let) rule.

(Ref-read) rule, for the occurrence $[!e^{p_1}]^{p'}$, evaluates the body e^{p_1} to a value, that must be a location $\updownarrow$, and reads the value of $\updownarrow$ in the store. The (Ref-read) rule also makes a lookup for the dependencies $\updownarrow$ is bound to in the dependency function w. As there could be multiple bindings for $\updownarrow$, in w, at different program points, we use the $uf_{\sqsubseteq_{w'}}$ function to get greatest binding of $\updownarrow$ with respect to the ordering $\sqsubseteq_{w'}$, and we also add the location occurrence $\updownarrow^{p'}$ to the set of locations.

(Ref-write) rule, for the occurrence $[e_1^{p_1} := e_2^{p_2}]^{p'}$, evaluate $e_1^{p_1}$ to a location $\updownarrow$ and $e_2^{p_2}$ to a value v, and binds $\updownarrow$ in the store sto to the value v. The dependency function is also updated with a new binding for $\updownarrow$ at the program point p'.

(Const)

$$env \vdash \left\langle c^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle c, sto, (w, \sqsubseteq_w), (\emptyset, \emptyset), p' \right\rangle$$

(Var)

$$env \vdash \left\langle x^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle v, sto, (w, \sqsubseteq_w), (L, V \cup \{x^{p'}\}), p' \right\rangle$$
Where $env(x) = v$, $x^{p''} = uf_{\sqsubseteq_w}(x, w)$, and $w(x^{p''}) = (L, V)$

(Let)

$$\frac{env \vdash \left\langle e_1^{p_1}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle v_1, sto_1, (w_1, \sqsubseteq_w^1), (L_1, V_1), p_1 \right\rangle \quad env[x \mapsto v_1] \vdash \left\langle e_2^{p_2}, sto_1, (w_2, \sqsubseteq_w^1), p_1 \right\rangle \rightarrow \left\langle v, sto', (w', \sqsubseteq'_w), (L, V), p_2 \right\rangle}{env \vdash \left\langle \left[\text{let } x\ e_1^{p_1}\ e_2^{p_2}\right]^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle v, sto', (w', \sqsubseteq'_w), (L, V), p' \right\rangle}$$
Where $w_2 = w_1[x^{p_1} \mapsto (L, V)]$

(Ref)

$$\frac{env \vdash \left\langle e^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle v, sto', (w', \sqsubseteq'_w), (L, V), p' \right\rangle}{env \vdash \left\langle \left[\text{ref } e^{p'}\right]^{p''}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle \updownarrow, sto'', (w'', \sqsubseteq'_w), (\emptyset, \emptyset), p'' \right\rangle}$$
Where $\updownarrow = next$, $sto'' = sto'[next \mapsto new(\updownarrow), \updownarrow \mapsto v]$, and
$w'' = w'[\updownarrow^{p'} \mapsto (L, V)]$

(Ref-read)

$$\frac{env \vdash \left\langle e^{p_1}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle \updownarrow, sto', (w', \sqsubseteq'_w), (L_1, V_1), p_1 \right\rangle}{env \vdash \left\langle [!e^{p_1}]^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle v, sto', (w', \sqsubseteq'_w), (L \cup L_1 \cup \{\updownarrow^{p''}\}, V \cup V_1), p' \right\rangle}$$
Where $sto'(\updownarrow) = v$, $\updownarrow^{p''} = uf_{\sqsubseteq'_w}(\updownarrow, w')$, and $w'(\updownarrow^{p''}) = (L, V)$

(Ref-write)

$$\frac{env \vdash \left\langle e_1^{p_1}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle \updownarrow, sto_1, (w_1, \sqsubseteq_w^1), (L_1, V_1), p_1 \right\rangle \quad env \vdash \left\langle e_2^{p_2}, sto_1, (w_1, \sqsubseteq_w^1), p_1 \right\rangle \rightarrow \left\langle v, sto_2, (w_2, \sqsubseteq_w^2), (L_2, V_2), p_2 \right\rangle}{env \vdash \left\langle \left[e_1^{p_1} := e_2^{p_2}\right]^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle (), sto', (w', \sqsubseteq'_w), (L_1, V_1), p' \right\rangle}$$
Where $sto' = sto_2[\updownarrow \mapsto v]$, $\updownarrow^{p''} = inf_{\sqsubseteq_w^2}(\updownarrow, w_2)$,
$w' = w_2[\updownarrow^{p'} \mapsto (L_2, V_2)]$, and $\sqsubseteq'_w = \sqsubseteq_w^2 \cup (p'', p')$

Fig. 2. Selected rules from the semantics

3 TYPE SYSTEM FOR DATA-FLOW ANALYSIS

This section will introduce the type system for data-flow analysis on the language presented in section 2. Similarly to the language, the type rules, types and other parts are based on [?]. The type system presented in this section is a type checker for local analysis of occurrences. The type checker assign types, presented in section 3.1, to occurrences given the basis, which will be presented in section 3.2, and using the type assignment, which is presented in section 3.3. Since the language contains local information as bindings, and global information as locations, the type checker should reflect this. Since locations are a semantic notation, we will present internal variables to represent locations in the type system, as νx, $\nu y \in \text{IVar}$ where IVar is the syntactic category for internal variables.

As references are not always bound to variables, as such the reference does not contain any alias information, the analysis provides alias information used for evaluating an occurrence. Here, we are going to introduce the basis for aliasing, as a partition of all variables and internal variables used in an occurrence. As such, it is possible to analyse, from the type information of occurrences, which aliases are actually used.

We also impose some restrictions on the type system, where the first restrictions is that references cannot be bound to abstractions. Since we do not introduce polymorphism, the use case for abstractions are reduced, as an abstraction cannot be used at multiple places. Consider the following occurrence:

$$(\texttt{let x } (\lambda \texttt{ y}.\texttt{y}^1)^2 \ (\texttt{x}^3 \ (\texttt{x}^4 \ 1^5)^6)^7)^8$$

To type the occurrences used in both places where we apply the abstractions, we type of the argument in the innermost application is empty, as it applies a constant. For the second, and outermost, application, the argument type must contain the occurrence x^4, as it were used to evaluate the value for the argument.

3.1 Types

We denote the set of types as Types, which are given by the following formation rules:

$$T ::= (\delta, \kappa) \mid T_1 \to T_2$$

Here, we introduce two types, the base type (δ, κ) and the abstraction type $T_1 \to T_2$. The idea is that an occurrence have the abstraction type if it represents an abstraction that takes an argument of type T_1 and returns a type of T_2. The base type represent all other values, where δ represent the set of occurrences, used to evaluate an occurrence, and κ represent the set of alias information. Here, if an occurrence have a type containing alias information, then it represent a location, where if the occurrence have a base type with alias information, then the occurrence must represents a reference. If the occurrence have the abstraction type where either T_1 or T_2 have are base types with alias information, then the abstraction either takes a reference as input or returns a reference.

Example 3.1. Consider the following occurrence:

$$(\texttt{let x } (3^1)^2 \ (\texttt{let y } (\texttt{ref x}^3)^4 \ (!\texttt{y})))$$

Here, we can type x with $(\emptyset, \emptyset)$ as x is bound to a constant and there a no variables or internal variables used. y can then be given the type $(\emptyset, \{x, \nu y\})$, as the reference construct ref creates a new reference, which y is then an alias to, e.g., y is bound to a location. Her νy represents the reference from ref, and can thus be given the type $(\{x^3\}, \emptyset)$, where νy is

bound to a constant, because of x, but the occurrence x^3 were used, so it should be part of the set of occurrences δ.

Since the type system approximates the occurrences used to evaluate an occurrence, we introduce two unions. The first union is a simple union that expects the types to be similar, that is, only the base types are allowed to be different.

Definition 3.1 (Type union). Let T_1 and T_2 be two types, then the type union, $\cup$, are as follows:

$$T_1 \cup T_2 = \begin{cases} \text{If } \ T_1 = (\delta, \kappa) \text{ and } T_2 = (\delta', \kappa') & \text{then } (\delta \cup \delta', \kappa \cup \kappa') \\ \text{else if } \ T_1 = T_1' \to T_1'' \text{ and } T_2 = T_2' \to T_2'' & \text{then } (T_1' \cup T_2') \to (T_1'' \cup T_2'') \end{cases}$$

The second type union, is to add additional type information to an arbitrary type. This type union is used to add an occurrence to a type, e.g., in the (Var) rule where the variable occurrence needs to be added to the type of that variable.

Definition 3.2 (Base type union). Let T be an type and (δ, κ) be a base type, then the union of these are as follows:

$$T \sqcup (\delta, \kappa) = \begin{cases} \text{If } \ T = (\delta', \kappa') & \text{then } (\delta \cup \delta', \kappa \cup \kappa') \\ \text{else if } \ T = T_1 \to T_2 & \text{then } T_1 \to (T_2 \sqcup (\delta, \kappa)) \end{cases}$$

3.2 Basis and type environment

Next, we will present the basis and type environment for the type system. The basis we are presenting here are assumptions used by the type checker, in addition to the assignment of types which are presented in section 3.3, where we are going to present a type base for aliasing and an approximated order of program points.

We will also introduce the type environment, which are similar to the environment and store used in the semantics, as the type environment keeps track of the type of variables and internal variables. As such, the type environment is also a approximation of the dependency function, as the purpose of the type system is to collect information about which occurrences are used and what alias information is used.

Similar to the lookup of the greatest binding for the dependency function, we are going to introduce an instantiation of the function from definition 2.13 for the type environment in respect to the basis for approximated order of program points.

We will then introduce the type base for aliasing, as a partition of variables and internal variables used in an occurrence.

Definition 3.3 (Type Base for aliasing). For an occurrence o, let var be the set of all variables and $ivar$ be the set of all internal variables in o. The type base $\kappa^0 = \{\kappa_1^0, \cdots, \kappa_n^0\}$ is then a partition of $var \cup ivar$, where $\kappa_i^0 \cap \kappa_j^0 = \emptyset$ for all $i \neq j$.

The idea behind the base for type alias κ_0 is to make a partition of the variables and internal variables used in an occurrence. This partition represents the assumption about which variables are actually an alias to internal variables. As such multiple variables can only belong to the same element $\kappa_0^i \in \kappa_0$, if there also exists an internal variable in κ_0^i.

Definition 3.4 (Approximated order of program points). An approximated order of program points Π is a pair, such that:

$$\Pi = (\mathrm{P}, \sqsubseteq_\Pi)$$

where

- P is the set of program points in an occurrence,
- $\sqsubseteq_\Pi \subseteq \mathrm{P} \times \mathrm{P}$, where

The approximated order of program points is an assumption about the order for program points for an occurrence o, as such, this approximation should be an approximation of the order that that can be derived from the semantics, presented in section 2.4, for o.

Definition 3.5 (Partial order of Π). Let $\Pi = (\mathrm{P}, \sqsubseteq_\Pi)$ be an approximated order of program points. We say that Π is a partial order if $\sqsubseteq_\Pi$ is a partial order.

Next, we will introduce the type environment:

Definition 3.6 (Type Environment). A type environment Γ is a partial function $\Gamma : \mathrm{Var}^P \cup \mathrm{IVar}^P \rightharpoonup \mathrm{Types}$

Definition 3.7 (Updating Type Environments). Let Γ be a type environment. We write $\Gamma[u^p : T]$, for an occurrence u^p, to denote the type environment Γ' where:

$$\Gamma'(y^{p'}) = \begin{cases} \Gamma(y^{p'}) & \text{if } y^{p'} \neq u^p \\ T & \text{if } y^{p'} = u^p \end{cases}$$

Similar to the lookup of dependencies in the semantics, we need to similarly define how to lookup in the type environment. As the type environment contains both local information, for local declarations, and global information, for references, both cases should be handled.

For local information we introduce, similarly to lookup in the dependency function, and instantiation of the function presented in definition 2.13. The lookup is for information in the type environment, over the relation between program points defined by the basis for approximated order of program points.

Definition 3.8. Let $u \in \mathrm{Var} \cup \mathrm{IVar}$, be either a variable or internal variable, Γ be a type environment, and Π be the approximated order of program points that is a partial order, then $uf_{\sqsubseteq_\Pi}$ is given by:

$$uf_{\sqsubseteq_\Pi}(u, \Gamma) = \inf\{u^p \in dom(\Gamma) \mid u^q \in dom(\Gamma).q \sqsubseteq_\Pi p\}$$

Where the lookup for global information needs to be handled differently as the language contains pattern matching, and as such, the language can contain different path of evaluation (where each pattern in the pattern matching construct introduces a new path). To handle the lookup of global information, we will first introduce the notion of p-chains as chains of program points with respect to the approximated order of program points, where the maximal program point is p. The idea of these p-chains is to describe the history behind an occurrence u^p, and can thus be used to describe what an internal variable depends on.

Definition 3.9 (p-chains). Let Π be an approximated order of program points, that is a partial order, and p be a program point. We then say that a p-chain, denoted as Π^*_p, is a maximal chain of with the maximal element p with the respect to the order Π. As such, any p-chain is a total order, where Π^*_p does not contain any pairs $(p, q) \in \sqsubseteq_\Pi$, where $p \neq q$, then $(p, q) \notin \sqsubseteq_{\Pi^*_p}$.

We also denote $\Pi^*_p \in \Pi$, if the p-chain Π^*_p can be be derived from Π. Since there can exists multiple paths in an occurrence, we define the set of all p-chains as follows:

Definition 3.10. Let Π be an approximated order of program points and p be a program point. We say that Υ_p is the set of all p-chains in Π.

Since Υ_p contains all p-chains in an approximated order of program points Π, with p as the maximal element, we can then define the function to lookup all greatest element less than or equal to p.

Definition 3.11. Let $u \in \text{Var} \cup \text{IVar}$, be either a variable or internal variable, Γ be a type environment, and Υ_p be a set of p-chains, then uf_{Υ_p} is given by:

$$uf_{\Upsilon_p}(u, \Gamma) = \bigcup_{\Pi_p^* \in \Upsilon_p} uf_{\Pi_p^*}(u, \Gamma)$$

The function, defined in definition 3.11, takes the union of the greatest binding, for an element, for each p-chain using the function defined in definition 3.8.

3.3 The type system

We will now present the judgement and type rules for the language, that is, how we assign types to occurrences.

The type judgement is defined as:

$$\Gamma, \Pi \vdash e^p : T$$

And should be read as: the occurrence e^p has type T, given the dependency bindings Γ and the approximated order of program points Π.

A highlight of type rules can be found in fig. 3, and all type rules can be found in ??.

(T-Const) rule, for occurrence c^p, is the simplest type rule, as there is nothing to track for constants, and as such it has the type $(\emptyset, \emptyset)$.

(T-Var) rule, for occurrence x^p, looks up the type for x in the type environment, by finding the greatest binding using definition 3.8, and add the occurrence x^p to the type.

(T-Let-1) rule, for occurrence $[\text{let } x\ e_1^{p_1}\ e_2^{p_2}]^p$, creates a local binding for a variable, with the type of $e_1^{p_1}$ that can be used in $e_2^{p_2}$. The (T-Let-1) rule assumes that the type of $e_1^{p_1}$ is a base type with alias information, i.e., $\kappa \neq \emptyset$. If this is the case, then $e_1^{p_1}$ must evaluate to a location, in the semantics. The other cases, when $e_1^{p_1}$ is not a base type with alias information, are handled by the (T-Let-2) rule. Since a pattern can be a variable, we updates the type environment with the type of e^p.

(T-Case) rule, for occurrence $[\text{case } e^p\ \tilde{\pi}\ \tilde{o}]^{p'}$, is an over-approximation of all cases in the pattern matching expression, by taking an union of the type of each case. Since the type of e^p is used to evaluate the pattern matching, we also add this type to the type of the pattern matching.

(T-Ref-read) rule, for occurrence $[!e^p]^{p'}$, is used to retrieve the type of references, where e^p must be a base type with alias information. Since the type system is an over-approximation, there can be multiple internal variables in κ and multiple occurrences we need to read from. As such we need to lookup for all internal variables and also possible for multiple program points. As such, we use the $uf_{Upsilon_{p'}}$ to lookup for all p'-chains.

Example 3.2 (Data-flow for abstractions). Consider the following occurrence for application:

$$((\lambda\ \text{y}\,.\,(\text{PLUS}\ \ 3^1\ \ \text{y}^2)^3)^4\ \ 5^5)^6$$

The derivation tree for the occurrence can be found in fig. 4. Here, we show two applications, for (T-App) and (T-App-const), where we create an abstraction that adds the constant 3 to the argument of the abstraction, and applying the constant 5 to the abstraction.

When typing the abstraction, we need too make an assumption about the parameter y and the body. As we are applying a constant to the argument, we can make an assumption that the type of the parameter should be $(\emptyset, \emptyset)$.

Based on this assumption for the type, we can then type the body of the abstraction. As the body is an application for a functional constant, (T-App-const), we take a union for the types of each argument.

Example 3.3 (Data-flow for references). Consider the following occurrence:

$$(\mathtt{let}\ \mathtt{x}\ (\mathtt{ref}\ 1^{1})^{2}\ (\mathtt{let}\ \mathtt{y}\ (\mathtt{x}^{3})\ (!\mathtt{x}^{4})^{5})^{6})^{7}$$

The derivation tree for the occurrence can be found in fig. 5. Here, we show the typing of references where we create a reference and create 2 aliases for it before reading from the reference. When typing the reference, it modifies the base type Γ with a new internal variable. From the type information, it is clear that only x and the internal variable νx is used.

(T-Const)

$$\frac{}{\Gamma, \Pi \vdash c^{p} : (\emptyset, \emptyset)}$$

(T-Var)

$$\frac{}{\Gamma, \Pi \vdash x^{p} : T \sqcup (\{x^{p}\}, \emptyset)}$$

$x^{p'} = uf_{\sqsubseteq_\Pi}(x, \Gamma)$, and $\Gamma(x^{p'}) = T$

(T-Let-1)

$$\frac{\Gamma, \Pi \vdash e_1^{p_1} : (\delta, \kappa) \quad \Gamma', \Pi \vdash e_2^{p_2} : T_2}{\Gamma, \Pi \vdash [\text{let } x\ e_1^{p_1}\ e_2^{p_2}]^{p} : T_2}$$

Where $\Gamma' = \Gamma[x^{p} : (\delta, \kappa \cup \{x\})]$ and $\kappa \neq \emptyset$

(T-Case)

$$\frac{\Gamma, \Pi \vdash e^{p} : (\delta, \kappa) \quad \Gamma', \Pi \vdash e_i^{p_i} : T_i \quad (1 \leq i \leq |\tilde{\pi}|)}{\Gamma, \Pi \vdash [\text{case } e^{p}\ \tilde{\pi}\ \tilde{o}]^{p'} : T \sqcup (\delta, \kappa)}$$

Where $e_i^{p_i} \in \tilde{o}$ and $s_i \in \tilde{\pi}$ $T = \bigcup_{i=1}^{|\tilde{\pi}|} T_i$, and
$\Gamma' = \Gamma[x^{p} : (\delta, \kappa)]$ if $s_i = x$

(T-Ref-read)

$$\frac{\Gamma, \Pi \vdash e^{p} : (\delta, \kappa)}{\Gamma, \Pi \vdash [!e^{p}]^{p'} : T \sqcup (\delta \cup \delta', \emptyset)}$$

Where $\kappa \neq \emptyset$, $\delta' = \{vx^{p'} \mid vx \in \kappa\}$, $vx_1, \cdots, vx_n \in \kappa$.
$\{vx_1^{p_1}, \cdots, vx_1^{p_m}\} = uf_{\Upsilon_{p'}}(vx_1, \Gamma), \cdots, \{vx_n^{p'_1}, \cdots, vx_n^{p'_s}\} = uf_{\Upsilon_{p'}}(vx_n, \Gamma)$, and
$T = \Gamma(vx_1^{p_1}) \cup \cdots \cup \Gamma(vx_1^{p_m}) \cup \cdots \cup \Gamma(vx_n^{p'_1}) \cup \cdots \cup \Gamma(vx_n^{p'_s})$

Fig. 3. Selected rules from the type system

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma', \Pi \vdash 3^1 : (\emptyset, \emptyset)}\,(\text{T-Const}) \quad \dfrac{}{\Gamma', \Pi \vdash y^2 : (\{y^2\}, \emptyset)}\,(\text{T-Var})}{\Gamma', \Pi \vdash [PLUS\ 3^1\ y^2]^3 : (\{y^2\}, \emptyset)}\,(\text{T-App-const})}{\Gamma, \Pi \vdash [\lambda\, y.(PLUS\ 3^1\ y^2)^3]^4 : (\emptyset, \emptyset) \to (\{y^2\}, \emptyset)}\,(\text{T-Abs}) \quad \dfrac{}{\Gamma, \Pi \vdash [5^5]^8 : (\emptyset, \emptyset)}\,(\text{T-Const})}{\Gamma, \Pi \vdash [(\lambda\, y.(PLUS\ 3^1\ y^2)^3)^4\ 5^5]^6 : (\{y^2\}, \emptyset)}\,(\text{T-Let})$$

Where $\Gamma' = \Gamma[y^{p_0} : (\emptyset, \emptyset)]$

Fig. 4. Abstraction type example

$$\dfrac{\dfrac{\dfrac{}{\Gamma, \Pi \vdash 1^1 : (\emptyset, \emptyset)}\,(\text{T-Const})}{\Gamma, \Pi \vdash [ref\ 1^1]^2 : (\emptyset, \{\nu x\})}\,(\text{T-Ref}) \quad \dfrac{\dfrac{}{\Gamma', \Pi \vdash x^3 : (\{x^3\}, \{\nu x\})}\,(\text{T-Var}) \quad \dfrac{\dfrac{}{\Gamma'', \Pi \vdash x^4 : (\{x^4, \nu x^5\}, \{\nu x\})}\,(\text{T-Var})}{\Gamma'', \Pi \vdash [!x^4]^5 : (\{x^4, \nu x^5\}, \emptyset)}\,(\text{T-Ref-read})}{\Gamma', \Pi \vdash [let\ y\ (x^3)\ (!x^4)^5]^6 : (\{x^4, \nu x^5\}, \emptyset)}\,(\text{T-Let})}{\Gamma, \Pi \vdash [let\ x\ (ref\ 1^1)^2\ (let\ y\ (x^3)\ (!x^4)^5)^6]^7 : (\{x^4, \nu x^5\}, \emptyset)}\,(\text{T-Let})$$

Where $\Gamma = \Gamma[\nu x^2 \mapsto (\emptyset, \emptyset)]$, $\Gamma' = \Gamma[x^2 \mapsto (\emptyset, \{\nu x\})]$, and $\Gamma'' = \Gamma'[y^3 \mapsto (\{x^3\}, \{\nu x\})]$

Fig. 5. Reference type example

4 SOUNDNESS

We will now show the soundness of the type system, i.e., the type of an occurrence correspond to the dependencies and the alias information from the semantics. To show that the type system is sound, we will first introduce the type rules for values and the relation between the semantics and the type system. After that, we will present some properties in the semantics and type system that are used in the soundness proof. And lastly, we will show the soundness of the type system.

4.1 Type rules for values

For the sake of proving the type system, we present type rules for the values presented int section 2.2, where the type rules is given in fig. 6.

As the values for closures and recursive closures contains an environment, from where they where declared, as such, before introducing the type rules for values we will present the notion for well-typed environments.

Definition 4.1 (Environment judgement). Let $v_1, \cdots, v_n$ be values such that $\Gamma, \Pi \vdash v_i : T_i$, for $1 \leq i \leq n$. Let env be an environment given by $env = [x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$, Γ be a type environment, and Π be the approximated order of program points. We say that:

$$\Gamma, \Pi \vdash env$$

iff

- For all $x_i \in dom(env)$ then $\exists x_i^p \in dom(\Gamma)$ where $\Gamma(x_i^p) = T_i$ then

$$\Gamma, \Pi \vdash env(x_i) : T_i$$

In definition 4.1 we show the notion of well-typed environments, which states that: given the type of all values, T_i, for all variables, x_i, bound in the env, then there exists an occurrence of x in Γ, where the type from looking up for that occurrences is is T_i. As such, we know that all bindings x in env also have a type in Γ from when x were declared.

(Constant) type rule differs from the rule (T-Const), since most occurrences can evaluate to a constant and as such we know that its type should be a base type. Since constants can depend on other occurrences, we know that δ can be non-empty, but since constants are not locations, we also know that it cannot contain alias information, and as such κ should be empty.

(Location) type rule represents locations, where we know that it must be a base type. Since locations can depend on other occurrences, we know that δ can be non-empty. As locations can contains alias information, and that a location is considered to always be an alias to itself, we know that κ can never be empty, as it should always contain an internal variable.

(Closure) type rule represents abstraction, and as such we know that it should have the abstraction type, $T_1 \to T_2$, where we need to make an assumption about the argument type T_1. Since a closure contains the parameter, body, and the environment for an abstraction from when it were declared, we also need to handle those part in the type rule. The components of the closure is handled in the premises, where the environment must be well-typed. We also type the body of the abstraction, where we know that we need to update the type environment with the type T_1 for its parameter, Where we type the body with T_2.

(Recursive closure) type rules is similar to the (Closure) rule, but since this is a recursive closure, we additionally need to update the type environment with the name of the recursive binding to the type of the abstraction.

(Unit) type rule simply have the base type, as it is not an abstraction and it also cannot have alias information. As the unit value is introduced from writing to references, $o = [o_1 := o_2]^p$, we know that from the type rule (Ref-write) that the dependencies from the occurrence o should also contain the set of occurrences. As such, the (Unit) rule also contains a set of occurrences, δ.

(Constant)

$$\frac{}{\Gamma, \Pi \vdash c : (\delta, \emptyset)}$$

(Location)

$$\frac{}{\Gamma, \Pi \vdash \updownarrow : (\delta, \kappa)}$$

Where $\kappa \neq \emptyset$

(Closure)

$$\frac{\Gamma, \Pi \vdash env \quad \Gamma[x^p : T_1], \Pi \vdash e^{p'} : T_2}{\Gamma, \Pi \vdash \left\langle x^p, e^{p'}, env \right\rangle^{p''} : T_1 \to T_2}$$

(Recursive closure)

$$\frac{\Gamma, \Pi \vdash env \quad \Gamma[x^p : T_1, f^{p'} : T_1 \to T_2], \Pi \vdash e^{p''} : T_2}{\Gamma, \Pi \vdash \left\langle x^p, f^{p'}, e^{p''}, env \right\rangle^{p_3} : T_1 \to T_2}$$

(Unit)

$$\frac{}{\Gamma, \Pi \vdash () : (\delta, \emptyset)}$$

Fig. 6. Type rules for values

4.2 Agreement

This section introduces the agreement between the type system and the semantics, where we will present the relation between the binding models in the type system and semantics, and show the relation between them.

We will first introduce the agreement between the binding models, i.e., show how the type environment and approximated order of program points relate to the environment, store, and dependency function. Then we will show the type agreement, i.e., show the conditions

for when a type agrees with the semantics. As such, the type agreement needs to show when the dependencies agrees and alias if the alias information agrees with the basis.

The first agreement we present is the environment agreement, which ensures that that the type environment and approximated order of program points are a good approximation of the binding model in the semantics, i.e., for the environment env, store sto, dependency function w, and the relation of program points over w.

Here env, sto, and w contains information for en evaluation in the semantics, either before or after an evaluation. The type environment Γ contains the local information for variable bindings and global information for internal variables, and the approximated order of program points Π is an approximation of all program points in an occurrence.

Definition 4.2 (Environment agreement). Let $(w, \sqsubseteq_w)$ be a pair containing the dependency function and a relation over it, env be an environment, sto be the a store, Γ be a type environment, and Π be an approximated program point order. We say that:

$$(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$$

if

(1) $\forall x \in dom(env).(\exists x^p \in dom(w)) \wedge (x^p \in dom(w) \Rightarrow \exists x^p \in dom(\Gamma))$
(2) $\forall x^p \in dom(w).x^p \in dom(\Gamma) \Rightarrow env(x) = v \wedge w(x^p) = (L, V) \wedge \Gamma(x^p) = T.$
$(env, v, (w, \sqsubseteq_w), (L, V)) \models (\Gamma, T)$
(3) $\forall \updownarrow \in dom(sto).(\exists \updownarrow^p \in dom(w)) \wedge (\exists vx.\forall p \in \{p' \mid \updownarrow^{p'} \in dom(w)\} \Rightarrow vx^p \in dom(\Gamma))$
(4) $\forall \updownarrow^p \in dom(w).\exists vx^p \in dom(\Gamma) \Rightarrow w(\updownarrow^p) = (L, V) \wedge \Gamma(vx^p) = T.(env, \updownarrow, (w, \sqsubseteq_w), (L, V)) \models T$
(5) if $p_1 \sqsubseteq_w p_2$ then $p_1 \sqsubseteq_\Pi p_2$
(6) $\forall \updownarrow^p \in dom(w).\exists vx^p \in dom(\Gamma) \Rightarrow \exists p' \in \mathrm{P}.uf_{\sqsubseteq_w}(\updownarrow, w) \in uf_{\Upsilon_{p'}}(vx, \Gamma)$

The idea behind the environment agreement is that we need to make sure that semantics and type system talks about the same, i.e., if the dependencies in the semantics is also captured in the type environment, the alias information is captured, that Π is a good approximation, in respect to w, and the p-chains captures the global occurrence. As such, the type environment focuses on three areas: 1) local information variables, 2) the global information for references, and 3) the approximated order of program points. It should be noted at the agreement only relates the information known by env, sto, and w.

1) The agreement for local information only relates the information currently known by env, and that the information known by w and Γ agrees, in respect to definition 4.3. This is ensured by 1) and 2).
2) We similarly handles agreement for the global information known, which is ensured by 3) and 4). Since Γ contains the global information for references, we require that there exists a corresponding internal variable to the currently known locations, by comparing them by program points. We also ensures that the dependencies from a location occurrence agrees with the type of a corresponding internal variable occurrence, in respect to definition 4.3.
3) We also needs to ensure that Π is a good approximation of the order $\sqsubseteq_w$ and the greatest binding function for p-chains ensures that we always get the necessary reference occurrences. 5) ensures that if an order is defined in $\sqsubseteq_w$, then Π also agrees on this order.
For 6), we need to ensure that for any location currently known the exists a corresponding internal variable where, getting the greatest binding of this occurrence, $\updownarrow^p$,

then there exists a program point p', such that looking up all greatest bindings for the p'-chain, there exists an internal variable occurrence that corresponds to $\updownarrow^p$.

With the environment agreement defined, we can present the type agreement. As the type can be abstractions and base types, with or without alias information, we have different requirements for handling them, as such we relate each requirement to a value Here, the idea is that if the value is a location, then we check that both the set of occurrences agrees with the dependency pair, presented in definition 4.4, and check if the alias information agrees with the semantics, definition 4.5. If the value is not a location, then the type can either be an abstraction type or base type. For the base type, we check that the agreement between the set of occurrences and the dependency pair agrees. If the type is an abstraction, then we check that T_2 agrees with binding model. We are only concerned about the return type T_2 for abstractions, since if the argument parameter is used in the body of the abstraction, then the dependencies would already be part of the return type.

Definition 4.3 (Type agreement). Let env be an environment, w be a dependency function, $\sqsubseteq_w$ be a relation over w, (L, V) be a dependency pair, and T be a type. We say that:

$$(env, v, (w, \sqsubseteq_w), (L, V)) \models (\Gamma, T)$$

iff

- $v \neq \updownarrow$ and $T = T_1 \to T_2$:
  - $(env, v, (w, \sqsubseteq_w), (L, V)) \models (\Gamma, T_2)$
- $v \neq \updownarrow$ and $T = (\delta, \kappa)$:
  - $(env, (L, V)) \models \delta$
- $v = \updownarrow$ then $T = (\delta, \kappa)$ where:
  - $(env, (L, V)) \models \delta$
  - $(env, (w, \sqsubseteq_w), v) \models (\Gamma, \kappa)$

Definition 4.4 (Dependency agreement). Let env be an environment, (L, V) be a dependency pair, and δ be a set of occurrences. We say that:

$$(env, (L, V)) \models \delta$$

if

- $V \subseteq \delta$,
- For all $\updownarrow^p \in L$ where $env^{\updownarrow} \neq \emptyset$, we then have $\{x \in dom(env) \mid env(x) = \updownarrow\} \subseteq \kappa_i^0$ for a $\kappa_i^0 \in \delta$
- For all $\updownarrow^p \in L$ where $env^{\updownarrow} = \emptyset$ then there exists a $\kappa_i^0 \in \delta$ such that $\kappa_i^0 \subseteq \text{IVar}$

The dependency agreement, defined in definition 4.4, ensures that δ at leas contains all information from the dependency pair.

Definition 4.5 (Alias agreement). Let env be an environment, w be a pair of a dependency function, $\sqsubseteq_w$ be a relation over w, $\updownarrow$ be a location, and κ be an alias set. We say that:

$$(env, (w, \sqsubseteq_w), \updownarrow) \models (\Gamma, \kappa)$$

if

- $\exists \updownarrow^p \in dom(w).vx^p \in dom(\Gamma) \Rightarrow vx \in \kappa$
- $env^{-1}(\updownarrow) \neq \emptyset . \exists \kappa_i^0 \in \kappa^0 \Rightarrow (env^{-1}(\updownarrow) \subseteq \kappa_i^0) \wedge (\exists \updownarrow^p \in dom(w).vx^p \in dom(\Gamma) \Rightarrow vx \in \kappa_i^0 \wedge vx \in \kappa)$
- $env^{-1}(\updownarrow) = \emptyset . \exists \kappa_i^0 \in \kappa^0 \Rightarrow (\exists \updownarrow^p \in dom(w).vx^p \in dom(\Gamma) \Rightarrow vx \in \kappa_i^0 \wedge vx \in \kappa)$

The alias agreement, defined in definition 4.5, ensures that the alias information in κ is also known the environment. To do this, we ensure that if there exists alias information in the environment env, then there exists an alias base $\kappa_i^0 \in \kappa^0$ such that the currently know alias information known in in env is a subset of κ_i^0, and that there exists a $vx \in \kappa$, such that $vx \in \kappa_i^0$. If there is no currently known alias information, we simply check that there exists a corresponding internal variable, that is part of an alias base.

4.3 Properties

Before we present the soundness proof, we will first present some properties about the semantics and type system. The first property we present is for the dependency function,since the dependency function is global, and as such they can contain side effects after an evaluation. This property states that if any new variable bindings is introduced to the dependency function, by evaluating an occurrence e^p, those variables are not free in e^p.

Lemma 4.1 (History). Suppose e^p is an occurrence, that

$$env \vdash \left\langle e^p, sto, (w, \sqsubseteq_w), p' \right\rangle \rightarrow \left\langle v, sto', (w', \sqsubseteq'_w), (L, V), p'' \right\rangle$$

and $x^{p_1} \in dom(w')\backslash dom(w)$. Then $x \notin fv(e^p)$

The proof for lemma 4.1 can be found in ??.

The second property is the strengthening of the type environment, which states that if there is a binding the type environment, used to type an occurrence e^p, and the variables is not free in e^p then the binding can be removed.

Lemma 4.2 (Strengthening). If $\Gamma[x^{p'} : T'], \Pi \vdash e^p : T$ and $x \notin fv(e^p)$, then $\Gamma, \Pi \vdash e^p : T$

The proof for lemma 4.2 can be found in ??.

With history, lemma 4.1, and strengthening, lemma 4.2, defined we can then present the soundness theorem. This theorem compares the semantics, for an occurrence, to a type rule that concludes this occurrence. Since we are interested in, if the type system is a sound approximation of the semantics, we need to make sure that an evaluation of an occurrence, and the type for the occurrence agrees. As such, we assume that the type environment and approximated order of program points are in an agreement with the binding models in the semantics, and we also assume that the environment is well-typed.

Based on these assumptions, we then need to make sure that, after an evaluation, we are still in agreement, we can type the value, and the type is in agreement with the semantics.

Theorem 4.3 (Soundness of type system). Suppose $e^{p'}$ is an occurrence where

- $env \vdash \left\langle e^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle v, sto', (w', \sqsubseteq'_w), (L, V), p'' \right\rangle$,
- $\Gamma, \Pi \vdash e^{p'} : T$
- $\Gamma, \Pi \vdash env$
- $(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$

Then we have that:

- $\Gamma, \Pi \vdash v : T$
- $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$
- $(env, (w', \sqsubseteq'_w), v, (L, V)) \models (\Gamma, T)$

Proof. The proof proceeds by induction on the height of the derivation tree for

$$env \vdash \left\langle e^{p'}, sto, \psi, p \right\rangle \rightarrow \left\langle v, sto', \psi', (L, V), p'' \right\rangle$$

We will only show the proof of four rules here, for (Var), (Case), (Ref), and (Ref-write), the full proof can be found in ??.

(Var) Here $e^{p'} = x^{p'}$, where

$$\frac{}{env \vdash \langle x^{p'}, sto, (w, \sqsubseteq_w), p\rangle \rightarrow \langle v, sto, (w, \sqsubseteq_w), (L, V \cup \{x^{p'}\}), p'\rangle}$$

Where $env(x) = v$, $x^{p''} = uf_{\sqsubseteq_w}(x, w)$, and $w(x^{p''}) = (L, V)$

And from our assumptions, we have:

- $\Gamma, \Pi \vdash x^{p'} : T$
- $\Gamma, \Pi \vdash env$
- $(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$

To type the occurrence $x^{p'}$ we use the rule (T-Var):

$$\frac{}{\Gamma, \Pi \vdash x^{p} : T \sqcup (\{x^{p}\}, \emptyset)}$$

Where $x^{p''} = uf_{\sqsubseteq_\Pi}(x, \Gamma)$, $\Gamma(x^{p''}) = T$.

We need to show that 1) $\Gamma, \Pi \vdash c : T$, 2) $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$, and 3) $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$.

1) Since, from our assumption, we know that $\Gamma, \Pi \vdash env$, we can then conclude that $\Gamma, \Pi \vdash v : T$
2) Since there are no updates to sto and $(w, \sqsubseteq_w)$, we then know from our assumptions that $(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$ holds after an evaluation.
3) Since there are no updates to sto' and $(w', \sqsubseteq'_w)$, that (L, V) is a result from looking up $x^{p''}$ in $(w, \sqsubseteq_w)$, and the type T is from to looking up $x^{p''}$ in Γ, we then know that $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$. Due to definition 4.3 we can conclude that:

$$(env, v, (w', \sqsubseteq'_w), (L, V \cup \{x^{p''}\})) \models (\Gamma, T \sqcup \{x^{p''}\})$$

(Case) Here $e^{p'} = \left[\text{case } e^{p''}\ \tilde{\pi}\ \tilde{o}\right]^{p'}$, where

$$\frac{\begin{array}{c} env \vdash \langle e^{p''}, sto, (w, \sqsubseteq_w), p\rangle \rightarrow \langle v_e, sto'', (w'', \sqsubseteq''_w), (L'', V''), p''\rangle \\ env[env'] \vdash \left\langle e_j^{p_j}, sto'', (w''', \sqsubseteq''_w), p''\right\rangle \rightarrow \langle v, sto', (w', \sqsubseteq'_w), (L', V'), p_i\rangle \end{array}}{env \vdash \left\langle \left[\text{case } e^{p''}\ \tilde{\pi}\ \tilde{o}\right]^{p'}, sto, (w, \sqsubseteq_w), p\right\rangle \rightarrow \langle v, sto', (w', \sqsubseteq'_w), (L, V), p'\rangle}$$

Where $match(v_e, s_i) = \perp$ for all $1 \leq u < j \leq |\tilde{\pi}|$, $match(v_e, s_j) = env'$, and $w''' = w''[x \mapsto (L'', V'')]$ if $env' = [x \mapsto v_e]$ else $w''' = w''$

And from our assumptions, we have that:

- $\Gamma, \Pi \vdash \left[\text{case } e^{p''}\ \tilde{\pi}\ \tilde{o}\right]^{p'} : T$,
- $\Gamma, \Pi \vdash env$
- $(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$,

To type $[\text{case } e^{p''}\ \tilde{\pi}\ \tilde{o}]^{p'}$ we need to use the (T-Case) rule, where we have:

$$\text{(T-Case)}\quad \frac{\Gamma, \Pi \vdash e^{p} : (\delta, \kappa) \qquad \Gamma', \Pi \vdash e_i^{p_i} : T_i \quad (1 \le i \le |\tilde{\pi}|)}{\Gamma, \Pi \vdash [\text{case } e^{p}\ \tilde{\pi}\ \tilde{o}]^{p'} : T}$$

Where $T = T' \sqcup (\delta, \kappa)$, $T' = \bigcup_{i=1}^{|\tilde{\pi}|} T_i$, $e_i^{p_i} \in \tilde{o}$ and $s_i \in \tilde{\pi}$, and $\Gamma' = \Gamma[x^{p} : (\delta, \kappa)]$ if $s_i = x$.
We must show that 1) $\Gamma, \Pi \vdash v : T$, 2) $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$, and
3) $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$.
To conclude, we first need to show for the premises, where due to our assumption and from the first premise, we can use the induction hypothesis to get:

- $\Gamma, \Pi \vdash v_e : (\delta, \kappa)$,
- $(env, sto'', (w'', \sqsubseteq''_w)) \models (\Gamma, \Pi)$,
- $(env, v, (w'', \sqsubseteq''_w), (L, V)) \models (\Gamma, (\delta, \kappa))$

Since in the rule (T-Case) we take the union of all patterns, we can then from the second premise:

- $\Gamma, \Pi \vdash v : T_j$,
- $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$,
- $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T_j)$

If we have a) $\Gamma', \Pi \vdash env[env']$ and b) $(env[env'], sto'', (w''', \sqsubset''_w)) \models (\Gamma', \Pi)$, we can then conclude the second premise by our induction hypothesis.

a) We know that either we have $\Gamma' = \Gamma[x \mapsto (\delta, \kappa)]$ and $env[x \mapsto v_e]$ if $s_j = x$, or $\Gamma' = \Gamma$ and env if $s_j \neq x$.
   - if $s_j \neq x$: Then we have $\Gamma, \Pi \vdash env$
   - if $s_j = x$: Then we have $\Gamma[x \mapsto (\delta, \kappa)], \Pi \vdash env[x \mapsto v_e]$, which hold due to the first premise.

b) We know that either we have $\Gamma' = \Gamma[x \mapsto (\delta, \kappa)]$ and $env[x \mapsto v_e]$ if $s_j = x$, or $\Gamma' = \Gamma$ and env if $s_j \neq x$.
   - if $s_j \neq x$: then we have $(env, sto'', (w'', \sqsubset''_w)) \models (\Gamma, \Pi)$.
   - if $s_j = x$: then $(env[x \mapsto v_e], sto'', (w''', \sqsubset''_w)) \models (\Gamma[x \mapsto (\delta, \kappa)], \Pi)$, since we know that $(env, sto'', (w'', \sqsubset''_w)) \models (\Gamma, \Pi)$, we only need to show for x. Since we have $x \in dom(env)$, $x^{p_j} \in dom(w''')$ and $x^{p_j} \in dom(\Gamma')$ and due to the first premise, we know that $(env[x \mapsto v_e], sto'', (w''', \sqsubset''_w)) \models (\Gamma[x \mapsto (\delta, \kappa)], \Pi)$.

Based on a) and b) we can then conclude:

1) Since $\Gamma', \Pi \vdash v : T_j$, then we also must have $\Gamma', \Pi \vdash v : T$, since T only contains more information than T_j.
2) By the second premise, lemma 4.1, and lemma 4.2, we can then get

$$(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$$

3) Due to 1), 2), a), and b) we can then conclude that

$$(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$$

(Ref-read) Here $e^{p'} = [!e_1^{p_1}]^{p'}$, where

(Ref-read)

$$\frac{env \vdash \left\langle e^{p_1}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle \updownarrow, sto', (w', \sqsubseteq'_w), (L_1, V_1), p_1 \right\rangle}{env \vdash \left\langle [!e^{p_1}]^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle v, sto', (w', \sqsubseteq'_w), (L \cup L_1 \cup \{\updownarrow^{p''}\}, V \cup V_1), p' \right\rangle}$$

Where $sto'(\updownarrow) = v$, $\updownarrow^{p''} = uf_{\sqsubseteq'_w}(\updownarrow, w')$, and $w'(\updownarrow^{p''}) = (L, V)$

And from our assumptions, we have that:
- $\Gamma, \Pi \vdash [!e_1^{p_1}]^{p'} : T$,
- $\Gamma; \Pi \vdash env$
- $(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$,

To type $[!e_1^{p_1}]^{p'}$ we need to use the (T-Ref-read) rule, where we have:

(T-Ref-read)

$$\frac{\Gamma, \Pi \vdash e^{p} : (\delta, \kappa)}{\Gamma, \Pi \vdash [!e^{p}]^{p'} : T \sqcup (\delta \cup \delta', \emptyset)}$$

Where $\kappa \neq \emptyset$, $\delta' = \{vx^{p'} \mid vx \in \kappa\}$, $vx_1, \cdots, vx_n \in \kappa$.
$\{vx_1^{p_1}, \cdots, vx_1^{p_m}\} = uf_{\Upsilon_{p'}}(vx_1, \Gamma), \cdots, \{vx_n^{p'_1}, \cdots, vx_n^{p'_s}\} = uf_{\Upsilon_{p'}}(vx_n, \Gamma)$, and
$T = \Gamma(vx_1^{p_1}) \cup \cdots \cup \Gamma(vx_1^{p_m}) \cup \cdots \cup \Gamma(vx_n^{p'_1}) \cup \cdots \cup \Gamma(vx_n^{p'_s})$.
We must show that (1) $\Gamma, \Pi \vdash v : T$, (2) $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$, and
(3) $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$.
To conclude, we first need to show for the premises, where due to our assumption and from the premise, we can use the induction hypothesis to get:
- $\Gamma, \Pi \vdash \updownarrow : (\delta, \kappa)$,
- $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$,
- $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, (\delta', \kappa'))$

Due to $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$ and $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, (\delta', \kappa'))$, and due to our assumptions, we can conclude that:
(1) $\Gamma, \Pi \vdash v : T$,
(2) $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$,
(3) $(env, v, (w', \sqsubseteq'_w), (L \cup \{\updownarrow^{p''}\}, V)) \models (\Gamma, T \sqcup (\delta \cup \delta', \emptyset))$

□