Modular, parsing-based, flow analysis of dictionary data structures in scripting languages

David A. Schmidt*

Kansas State University, Manhattan, Kansas, USA

Abstract. We design and implement a modular, constant-propagation-like forwards flow analysis for a Python subset containing strings and dictionaries (hash tables). The analysis infers types of dictionaries and the functions and modules that use them. Unlike records and class-based objects, dictionaries are wholly dynamic, and we employ a domain of dictionary types that delineate which fields a dictionary must have. We have deliberately omitted unification-based inference and row variables to obtain the benefits of a forwards analysis that matches a programmer's intuitions. Nonetheless, to accommodate a modular analysis, the values of parameters and free (global) variables are represented by tokens to which are attached constraints. At link- and function-call-time, the constraints are matched against the actual values of arguments and global variables.

Finally, programmers are encouraged to use a BNF-like syntax to define the forms of data types employed in their scripts. The analysis uses the programmer-written BNF rules to "abstractly parse" program phrases and associate them with derivations possible from the programmer-defined grammars. A prototype of the system is under construction.

1 Introduction

Programmers like the dynamic data structures of scripting languages like Perl, Ruby, and Python. The dictionary (hash table) structure is particularly useful for data modelling. For example, one can model an empty, heterogenous binary tree by the string, `'nil'`, and a nonempty tree by the dictionary, `{'val': V, 'left':T1, 'right':T2}`, where `V` can be any phrase whatsoever and each `Ti` can be a subtree. By omitting the `'left'` and `'right'` fields, we model partial trees as well. Functions that process partial trees are simply written; see Figure 1.

The partial-tree data structure is difficult to define simply in a typed language. But such heterogeneous data structures are common in practice and are handled well by dynamically-typed scripting languages. Unfortunately, when a single script or a collection of script-coded modules grows above 5000 lines, loss of comprehension cancels the gain of simplicity. Further, the time spent debugging data-structure mismatch errors ("typing errors") becomes significant.

* das@ksu.edu. Supported by NSF CNS-0939431.

```
def count(t): # counts the values in tree t
  if t == 'nil': # empty tree ?
    return 0
  elif isinstance(t, dict): # a nonempty tree ?
    ans = 1
    if 'left' in t: ans = ans + count(t['left'])
    if 'right' in t: ans = ans + count(t['right'])
    return ans
```

Fig. 1. Function that counts nodes in a partial binary tree

As researchers have noted [2, 3, 8–11], an ideal solution would be a minimally programmer-assisted, type-inference tool to check script modules for data-structure mismatches. This paper describes an approach that does dictionary checking and complements the work on record and class-based object checking [1, 4, 6, 12]

2 Related efforts

Two key previous efforts relevent to our work are *soft typing* [2] and *gradual typing* [8, 9].

Soft typing is an ML-like type inference which infers types from the type domain of primitive types, function types, true unions, and fixed points. Phrases that cannot be assigned a type are wrapped with a run-time cast. The standard implementation of soft typing encodes a union type as a record whose fields enumerate all the type constructors in the programming language; a field's value is set to *must* if its constructor is a component of the union type, and *must-not*, when it is certainly excluded. To define subtyping of unions, a field can also take on a *may* value. Remy's inference algorithm for records, paired with Huet's unification algorithm, are used to perform the inference and synthesis of recursive defined union types [6]. Soft typing is elegant but is not modular and does not scale well [3].

Gradual typing is a type inference for a type calculus that is augmented by a type *dynamic* ("?") or "unknown." The typing is named "gradual" because the programmer is encouraged to refine the program with more and more types to reduce occurrences of ?. Checking is performed up to the occurrences of ?, which are checked at run-time. This induces a type compatibility relationship, $\sim$, such that $int \sim ?$, $(int \rightarrow ?) \sim (? \rightarrow bool)$, etc. The definition of $\sim$ is compatible with type refinement (replacement of occurrences of ? by concrete types). Gradual typing has been defined for ML and Java subsets (whose class definitions aid type inference) [8, 9].

Two other recent efforts deserve mention: Furr, et al. [4] have defined a Ruby subset, DRuby, along with a type-inference algorithm. DRuby does not support dictionaries, and it depends on class definitions to check object creation and message passing. Jensen, et al. [5] have designed a precise flow analysis to detect

$$
\begin{array}{ll}
C \in Command & B \in BoolExpression \\
L \in LeftHandSide & I \in Identifier \\
E \in Expression & S \in StringConstant
\end{array}
$$

$$
\begin{array}{rcl}
C & ::= & L \texttt{ = } E \mid \texttt{if } B\texttt{:} C_1 \texttt{ else: } C_2 \mid \texttt{while } B\texttt{:} C \\
 & & \mid \texttt{def } I_1(I_2)\texttt{: } C \mid \texttt{global } I \mid \texttt{return } E \mid C_1\texttt{; } C_2 \\
L & ::= & I \mid E\texttt{[}I\texttt{]} \\
E & ::= & S \mid stringop\ E \mid \{E_i : E'_i\}_{0 \le i < m} \mid L \mid L\texttt{(}E\texttt{)} \\
B & ::= & E_1 \texttt{ == } E_2 \mid E \texttt{ in } L \mid \texttt{isinstance(}L\texttt{,str)} \\
 & & \mid \texttt{isinstance(}L\texttt{,dict)} \mid \texttt{callable(}L\texttt{)}
\end{array}
$$

Fig. 2. Syntax of Python subset

errors in Javascript scripts, which use prototype objects that provide information similar to a class definition's.

Over a number of years, Felleisen and his colleagues have implemented and profiled several soft-typing and set-based analyses for module-based Scheme [3]. Their conclusions are that the methods do not scale well, are "brittle" (type errors easily arise) and do not provide good error diagnostics, and must be augmented by additional theory to accommodate modules [3]. Their latest efforts have moved towards type checking of Scheme augmented by programmer-written ML-like `datatype` definitions and annotated case-selector functions [3, 10, 11].

3 Script modules with dictionaries

Based on our experiences working with scripting languages and based on the research documented above, we learned there is value in having a tool assistant that can warn a programmer of mismatches in dictionary usage. We realize that is impossible to impose a static typing discipline on a script, so we search for a middle ground, where a modular, forwards, data-flow-analysis-based inference algorithm provides directives to the programmer where repairs might be necessary. For the reasons outlined by Felleisen [3], we employ an object-based, forwards, iterative data-flow constant-propagation algorithm as our tool's software architecture. The tool accepts programmer assistance supplied as generalized `datatype` definitions and parameter annotations.

We start our work with a Python subset, consisting of strings, dictionaries, conditionals, loops, and functions. Every Python script is itself a module, which is executed (*imported*) and then can be linked to other modules that use the script's public namespace (variable and function definitions). The source syntax is defined in Figure 2.

Figure 3 shows a small script with strings, dictionaries, and nested scopes. Python employs static scoping on its namespaces (environments), but declarations are unneeded — namespaces are dynamically filled, as when `f` is called in the example: `f`'s local namespace holds a binding for `z`, and its global names-

```
def f(z):
  global x
  if isinstance(y, dict): y[x] = z
  x = y
def g():
  y = 'cd'; f(y)
x = 'ab'; y = {}; g()
while x in y: # is x a field in dictionary y ?
  print y[x]; y = y[x]
```

Fig. 3. Sample script/module

pace, whose identity was fixed at `f`'s definition, is consulted for `x`. Since `y` cannot be found in `f`'s local namespace, the global namespace is searched for it. Note that global `y`'s value is *not* changed by calling `g`. Finally, there is a potential for generating an exception by evaluating the test, `x in y`, at the head of the while-loop, because `y` might take on a non-dictionary value. (This possibility will be detected by our analysis.) The example shows that Python's static scoping and its dictionaries pose problems in that both are modelled by namespaces that are dynamically filled with fieldname, value pairs. Also, fieldnames for dictionary lookups are computed at run-time, like array indexes, and dictionaries can be aliased.

Figures 4 and 5 give the language's denotational semantics. The semantics shows that a command's environment is a stack (list) of locations (addresses) of namespaces in the heap. The environment is statically calculated, but each namespace referencd by an environment is dynamically filled. The "$global$" token is a "patch" that allows a function to assign to a top-level variable. Note that primitive values (here, strings), dictionaries (namespaces) and closures can be stored in the heap and assigned to variable names.

4 Modular forwards constant-propagation analysis

The error-detection analysis is an abstract interpretation of the concrete semantics, where the main abstraction is upon namespaces:

$$AbsNS = \{(AbsStr : AbsDenotable)^*\}[!]$$
$$AbsStr = String \cup \{Str_{any}\}$$
$$AbsDenotable = SetOf\,(AbsStr + NSLocation + ClLocation)$$

That is, an abstract namespace is a dictionary of abstract-string, denotable-set, mappings with an optional ! suffix, which denotes "exactly". For example, the abstract namespace, $\{"a" : \{"mm", \ell_0\}, "b" : Str_{any}\}!$ denotes all namespaces that have exactly two fields, "a" and "b", where "a" denotes either string "mm" or location ℓ_0 and "b" maps to some string (represented by Str_{any}). Since "b"'s value is a singleton set, we omit the set braces to ease eye strain. If we remove the ! suffix, the abstract namespace denotes those namespaces that have *at least* the

Semantic domains:

$\rho \in Environment = NSLocation^*$ (stack of namespaces' locations)
$\sigma \in Heap = (NSLocation \to Namespace) \times (ClLocation \to Closure)$
$Namespace = Iden \to Denotable$, where $Iden \subseteq String$
$Denotable = String + NSLocation + ClLocation + "global"$
$Expressible = Denotable + "error"$
$Closure = Iden \times Command \times Environment$
$LValue = NSLocation \times Iden$
$Outcome = Heap + (Expressible \times Heap)$

Auxiliary functions:

$allocateNS : Denotable \times Heap \to NSLocation \times Heap$
$\quad allocateNS(v, \sigma) = (\ell_0, \sigma + [\ell_0 \mapsto v])$, where ℓ_0 is a fresh location
$allocateCl : Denotable \times Heap \to ClLocation \times Heap$ (similar)

$findNSof : Iden \times Environment \times Heap \to NSLocation + "none"$
$\quad findNSof(i, nil, \sigma) = "none"$
$\quad findNSof(i, \ell :: \rho', \sigma) = \mathit{if}\ i \in dom(\sigma(\ell))$
$\qquad \mathit{then\ if}\ \sigma(\ell) = "global"\ \mathit{then}\ last(\ell :: \rho')\ \mathit{else}\ \ell$
$\qquad\quad$ where $last(\ell_0 :: \ell_1 :: \cdots :: \ell_n) = \ell_n$
$\qquad \mathit{else}\ findNSof(i, \rho', \sigma)$

$lookup : LValue \times Heap \to Denotable + "error"$
$\quad lookup((\ell, i), \sigma) = \mathit{if}\ i \in \sigma(\ell)\ \mathit{then}\ \sigma(\ell)(i)\ \mathit{else}\ "error"$

$update : LValue \times Denotable \times Heap \to Heap$
$\quad update((\ell, i), v, \sigma) = \sigma + [\sigma(\ell) \mapsto \sigma(\ell) + [i : v]]$

$apply : (Heap \to Outcome) \times Outcome \to Outcome$
$\quad apply(f, \psi) = \mathit{if}\ \psi \in Heap\ \mathit{then}\ f(\psi)\ \mathit{else}\ \psi$

$\mathcal{C} : Command \to Environment \to Heap \to Outcome$

$$\mathcal{C}[\![L \texttt{ = } E]\!]\rho\sigma = \begin{array}{l} \mathit{let}\ \theta, \sigma' = \mathcal{L}[\![L]\!]\rho\sigma \\ \mathit{let}\ v, \sigma'' = \mathcal{E}[\![E]\!]\rho\sigma' \\ \mathit{in}\ update(\theta, v, \sigma'') \end{array}$$

$$\mathcal{C}[\![\texttt{if}\ B\ \texttt{:}C_1\ \texttt{else:}\ C_2]\!]\rho\sigma = \begin{array}{l} \mathit{let}\ b, \sigma' = \mathcal{B}[\![B]\!]\rho\sigma \\ \mathit{if}\ b\ \mathit{then}\ \mathcal{C}[\![C_1]\!]\rho\sigma'\ \mathit{else}\ \mathcal{C}[\![C_1]\!]\rho\sigma' \end{array}$$

$$\mathcal{C}[\![\texttt{while}\ B\ \texttt{do:}\ C]\!]\rho = f,$$
$$\text{where } f\sigma = \begin{array}{l} \mathit{let}\ b, \sigma' = \mathcal{B}[\![B]\!]\rho\sigma \\ \mathit{if}\ b\ \mathit{then}\ apply(f, \mathcal{C}[\![C]\!]\rho\sigma')\ \mathit{else}\ \sigma' \end{array}$$

$$\mathcal{C}[\![\texttt{def}\ I_1(I_2)\texttt{:}\ C]\!]\rho\sigma = \begin{array}{l} \mathit{let}\ \theta, \sigma_1 = \mathcal{L}[\![I_1]\!]\rho\sigma \\ \mathit{let}\ \ell_2, \sigma_2 = allocateNS(\{\}, \sigma_1) \\ \mathit{let}\ \ell_3, \sigma_3 = allocateCl((I_2, C, (\ell_0 :: \rho)), \sigma_2) \\ \mathit{in}\ update(\theta, \ell_3, \sigma_3) \end{array}$$

$$\mathcal{C}[\![\texttt{global}\ I]\!]\rho\sigma = update((head(\rho), I), "global", \sigma)$$

$$\mathcal{C}[\![\texttt{return}\ E]\!]\rho\sigma = \mathcal{E}[\![E]\!]\rho\sigma$$

$$\mathcal{C}[\![C_1\ \texttt{;}\ C_2]\!]\rho\sigma = apply(\mathcal{C}[\![C_2]\!]\rho, \mathcal{C}[\![C_1]\!]\rho\sigma)$$

Fig. 4. Semantics of Python subset

$$\mathcal{L} : LeftHandSide \to Environment \to Heap \to ((LValue + \text{"}error\text{"}) \times Heap)$$

$$\mathcal{L}[\![I]\!]\rho\sigma = \begin{array}{l} let\ \ell = findNSof(I, \rho, \sigma) \\ if\ \ell = \text{"}none\text{"}\ then\ ((head(\rho), I), \sigma)\ \ else\ ((\ell, I), \sigma) \end{array}$$

$$\mathcal{L}[\![E[I]]\!]\rho\sigma = \begin{array}{l} let\ \ell, \sigma' = \mathcal{E}[\![E]\!]\rho\sigma \\ if\ v \in NSLocation\ then\ ((v, I), \sigma')\ \ else\ (\text{"}error\text{"}, \sigma') \end{array}$$

$$\mathcal{E} : Expression \to Environment \to (Expressible \times Heap)$$

$$\mathcal{E}[\![\texttt{"abc"}]\!]\rho\sigma = \text{"}abc\text{"}, \sigma$$

$$\mathcal{E}[\![L]\!]\rho\sigma = let\ \theta, \sigma' = \mathcal{L}[\![L]\!]\rho\sigma\ \ in\ (lookup(\theta, \sigma'), \sigma')$$

$$\mathcal{E}[\![L(E)]\!]\rho\sigma = \begin{array}{l} let\ \ell, \sigma' = \mathcal{E}[\![L]\!]\rho\sigma \\ if\ \ell \notin ClLocation\ then\ (\text{"}error\text{"}, \sigma') \\ else\ let\ I, C, (\ell_0 :: \rho_0) = \sigma(\ell) \\ \quad let\ v, \sigma'' = \mathcal{E}[\![E]\!]\rho\sigma' \\ \quad in\ if\ v = \text{"}error\text{"}\ \ then\ (\text{"}error\text{"}, \sigma'') \\ \qquad else\ \mathcal{C}[\![C]\!](\ell_0 :: \rho)(update((\ell_0, I), v, \sigma'')) \end{array}$$

$$\mathcal{E}[\![\{\, E_i : E'_i \}_{0 \le i < m}]\!]\rho\sigma = \begin{array}{l} let\ \ell, \sigma_0 = allocateNS(\sigma) \\ foreach\ E_i, E'_i, 0 \le i < m, \\ \quad let\ \phi_i, \sigma'_i = \mathcal{E}[\![E_i]\!]\rho\sigma_i \\ \quad let\ v_i, \sigma''_i = \mathcal{E}[\![E'_i]\!]\rho\sigma'_i \\ \quad let\ \sigma_{i+1} = update((\ell, \phi_i), v_i, \sigma''_i) \\ in\ (\ell, \sigma_m) \end{array}$$

$$\mathcal{B} : BoolExpr \to Environment \to Heap \to (Boolean + \text{"}error\text{"}) \times Heap$$

$$\mathcal{B}[\![E\ \texttt{in}\ L]\!]\rho\sigma = \begin{array}{l} let\ v_1, \sigma_1 = \mathcal{E}[\![E]\!]\rho\sigma \\ let\ v_2, \sigma_2 = \mathcal{E}[\![L]\!]\rho\sigma_1 \\ if\ v_1 \notin String\ or\ v_2 \notin NSLocation\ then\ (\text{"}error\text{"}, \sigma_2) \\ \quad else\ (v_1 \in dom(\sigma_2(v_2)), \sigma_2) \end{array}$$

$$\mathcal{B}[\![\texttt{instanceof}(L, \texttt{dict})]\!]\rho\sigma = \begin{array}{l} let\ v, \sigma' = \mathcal{E}[\![L]\!]\rho\sigma \\ if\ v = \text{"}error\text{"}\ \ then\ (\text{"}error\text{"}, \sigma') \\ else\ (v \in NSLocation, \sigma') \end{array}$$

$$\mathcal{B}[\![\texttt{callable}(L)]\!]\sigma = \begin{array}{l} let\ v, \sigma' = \mathcal{E}[\![L]\!]\rho\sigma \\ if\ v = \text{"}error\text{"}\ \ then\ (\text{"}error\text{"}, \sigma')\ \ else\ (v \in ClLocation, \sigma') \end{array}$$

$$\mathcal{B}[\![E_1 \texttt{==} E_2]\!]\rho\sigma = \begin{array}{l} let\ v_1, \sigma_1 = \mathcal{E}[\![E_1]\!]\rho\sigma \\ let\ v_2, \sigma_2 = \mathcal{E}[\![E_2]\!]\rho\sigma_1 \\ if\ v_1\ or\ v_2 = \text{"}error\text{"}\ \ then\ (\text{"}error\text{"}, \sigma_2)\ \ else\ (v_1 \equiv v_2, \sigma_2) \\ \text{where}\ \equiv\ \text{defines equality on strings, "deep equality"} \\ \text{on namespaces, and } ClLocation \text{ equality on closures} \end{array}$$

Fig. 5. Semantics of Python subset, concl.

"a" and "b" fields with the values just described. Figure 6 defines the abstract patterns, their concretizations, and their partial orderings.

Abstract domains:

$$AbsStr = String \cup \{Str_{any}\}$$
$$AbsDenotable = Set\,Of\,(AbsStr + NSLocation + ClLocation)$$
$$AbsNS = \{(String : AbsDenotable)^*\}[!] + \{Str_{any} : AbsDenotable\}$$

Concretization functions on abstract strings, denotables, and namespaces:

$$\gamma : AbsStr \to \mathcal{P}(String)$$
$$\gamma(s) = \{s\}$$
$$\gamma(Str_{any}) = String$$

$$\gamma : AbsDenotable \to \mathcal{P}(Denotable)$$
$$\gamma\{v_i\}_{0\le i\le n} = \bigcup_{0\le i\le n} \gamma(v_i)$$
where $\gamma(\ell) = \{\ell\}$, for $\ell \in NSLocation \cup ClLocation$

$$\gamma : AbsNS \to \mathcal{P}(Namespace)$$
$$\gamma\{s_i : T_i\}_{0\le i<n} = \{d \in Namespace \mid \text{for every } s_i, 0 \le i < n,$$
there exists $s' \in dom(d)$ such that $s' \in \gamma(s_i)$ and $d(s') \in \gamma(T_i)\}$
$$\gamma\{s_i : T_i\}_{0\le i<n}! = \{d \in Namespace \mid dom(d) = \bigcup_{0\le i\le n} \gamma(s_i),$$
and for every $0 \le i < n, d(s_i) \in \gamma(T_i)\}$

Partial orderings: (reflexive, transitive closure of)

AbsString :
$s \sqsubseteq Str_{any}$, for all $s \in String$

AbsDenotable :
$T \sqsubseteq T'$ if for every $v \in T$ there exists $v' \in T'$ such that $v \sqsubseteq v'$

AbsNS: let $d_1 = \{s_i : v_i\}_{0\le i<m}$ and $d_2 = \{s'_j : v'_j\}_{0\le j<n}$:
$d_1[!] \sqsubseteq d_2$ if for every $s'_j \in dom(d_2), 0 \le j < n,$
there exists $s_i \in dom(d_1)$ such that $s_i \sqsubseteq s'_j$ and $d_1[s_i] \sqsubseteq d_2[s'_j]$
$d_1! \sqsubseteq d_2!$ if $dom(d_1) = dom(d_2)$ and for every $s'_j \in dom(d_2), d_1[s'_j] \sqsubseteq d_2[s'_j]$

Fig. 6. Abstract domains of strings and namespaces

The partial orderings are used by the flow-analysis algorithm to compute least fixed points — the key property is that $P_1 \sqsubseteq P_2$ implies $\gamma(P_1) \subseteq \gamma(P_2)$. Since the token, Str_{any}, denotes "any string at all", we have that "abc" $\sqsubseteq Str_{any}$, etc. An abstract denotable value is now a set of possible denotables, and two abstract denotable sets are compared by using the lower powerset ordering, e.g., $\{$"abc", $\ell_2\} \sqsubseteq \{Str_{any}, \ell_2, \ell_1\}$. Dictionaries are ordered by contravariance on the field names: $\{$"a" : $\{$"m", "n"$\}$, "b" : $\{\ell_0\}\}! \sqsubseteq \{$"$a$" : $\{Str_{any}\}$, "b" : $\{\ell_0$, "n"$\}\}!$ $\sqsubseteq \{$"a" : $\{Str_{any}\}$, "b" : $\{\ell_0$, "n"$\}\} \sqsubseteq \{$"b" : $\{\ell_0$, "n"$\}\} \sqsubseteq \{$"b" : $\{\ell_0, Str_{any}\}\}$ $\sqsubseteq \{Str_{any} : \{\ell_0, Str_{any}\}\} \sqsubseteq \{\}$.

Program and flow equations:

```
x = {}_α
while ...:
    x['a'] = {}_β
    x = x['a']
... x['a'] ...
```

$$
\begin{aligned}
P_0 &= initialHeap_{n_0} \\
P_1 &= assign\ x\ \{\}_\alpha\ P_0 \\
P_2 &= P_1 \sqcup P_4 \\
P_3 &= assign\ (index\ x\ \text{"}a\text{"})\ \{\}_\beta\ P_2 \\
P_4 &= assign\ x\ (index\ x\ \text{"}a\text{"})\ P_3 \\
P_5 &= \cdots (index\ x\ \text{"}a\text{"}) \cdots P_2
\end{aligned}
$$

At all program points, P_i, the environment has value, $[n_0]$, where n_0 is the address of the global namespace. The initial heap is $[n_0 \mapsto \{\}!] \times []$ — there are no variable bindings yet in n_0's namespace. The second heap component, [] (closure bindings) stays empty and is omitted hereon. The worklist solution to the flow equations goes as follows:

$$
\begin{aligned}
P_0 &= [n_0 \mapsto \{\}!] \\
P_1 &= [n_0 \mapsto \{x : \ell_\alpha\}!, \ell_\alpha \mapsto \{\}!] \\
P_2 &= [n_0 \mapsto \{x : \ell_\alpha\}!, \ell_\alpha \mapsto \{\}!] \\
P_3 &= [n_0 \mapsto \{x : \ell_\alpha\}!, \ell_\alpha \mapsto \{\text{"}a\text{"} : \ell_\beta\}!, \ell_\beta \mapsto \{\}!] \\
P_4 &= [n_0 \mapsto \{x : \ell_\beta\}!, \ell_\alpha \mapsto \{\text{"}a\text{"} : \ell_\beta\}!, \ell_\beta \mapsto \{\}!] \\
P_2 &= [n_0 \mapsto \{x : \{\ell_\alpha, \ell_\beta\}\}!, \ell_\alpha \mapsto \{\}, \ell_\beta \mapsto \{\}!] \\
P_3 &= [n_0 \mapsto \{x : \{\ell_\alpha, \ell_\beta\}\}!, \ell_\alpha \mapsto \{\text{"}a\text{"} : \ell_\beta\}, \ell_\beta \mapsto \{\text{"}a\text{"} : \ell_\beta\}!] \\
P_4 &= [n_0 \mapsto \{x : \ell_\beta\}!, \ell_\alpha \mapsto \{\text{"}a\text{"} : \ell_\beta\}, \ell_\beta \mapsto \{\text{"}a\text{"} : \ell_\beta\}!] \\
P_2 &= [n_0 \mapsto \{x : \{\ell_\alpha, \ell_\beta\}\}!, \ell_\alpha \mapsto \{\}, \ell_\beta \mapsto \{\}] \\
P_3 &= [n_0 \mapsto \{x : \{\ell_\alpha, \ell_\beta\}\}!, \ell_\alpha \mapsto \{\text{"}a\text{"} : \ell_\beta\}, \ell_\beta \mapsto \{\text{"}a\text{"} : \ell_\beta\}] \\
P_4 &= [n_0 \mapsto \{x : \ell_\beta\}!, \ell_\alpha \mapsto \{\text{"}a\text{"} : \ell_\beta\}, \ell_\beta \mapsto \{\text{"}a\text{"} : \ell_\beta\}] \\
P_2 &= [n_0 \mapsto \{x : \{\ell_\alpha, \ell_\beta\}\}!, \ell_\alpha \mapsto \{\}, \ell_\beta \mapsto \{\}] \\
&\text{(converges)}
\end{aligned}
$$

(To reduce eye strain, the brackets around singleton-set abstract denotables are omitted.)

Fig. 7. Sample program and its flow analysis

Using the abstract domains, we compute a forwards data-flow analysis, where each line in the source program generates one or more flow equations. The family of simultaneously defined flow equations are solved so that each equation computes to an environment (stack of $NSlocation$) and abstract heap of form $(NSlocation \to AbsNS) \times (CLlocation \to Closure)$. Since Python is statically scoped, the computation of the enviroments can be undertaken prior to the computation of the abstract heaps.

Figure 8 shows a small loop program, its flow equations, and the solution to the equations. Because the loop's iterations are unknown, so is the quantity of dictionaries allocated within the loop. For simplicity, we employ *summary nodes* [7, 13], ℓ_α and ℓ_β, to denote the dictionary objects allocated during the analysis. The heap at loop exit is summarized by $[n_0 \mapsto \{x : \{\ell_\alpha, \ell_\beta\}\}!, \ell_\alpha \mapsto \{\}, \ell_\beta \mapsto \{\}\}]$. That is, at loop exit, x (and no other variable) is definitely

defined in the global namespace and denotes a dictionary. The dictionary's value is overapproximated by $\{\}$, which means the dictionary's fields and values are uncertain; the dictionary might be empty. (Indeed, this is the case at run time.) The heap lets us analyze the reference to `x['a']` at point P_5 — a missing-field error might occur, since there is no guarantee that the field labelled `'a'` exists.

The primary purpose of the analysis is to detect operator-operand incompatibility errors and missing-field errors (*key errors*) in dictionary usage. The analysis looks superficially like type analyses for class-based object-oriented languages, but as the previous example shows, dictionaries pose new challenges in that fields can be added dynamically and field names for lookups can be computed at run-time.

5 Function-definition analysis

To make the analysis modular, we analyze a function definition with an abstract heap that contains symbolic constants, called *tokens*, for the values of the function's free variables, namely, the function's formal parameters and nonlocal variables.

While the function body is analysed, references to the free variables generate *conditional constraints* on the corresponding tokens. The constraints are included along with the function's output image of the abstract heap, as the function's abstract denotation — a summary template. At function-call time, the template is compared to the actual-parameter arguments and global-variable values in the call-time heap for a compatibility check ("type check"). If the constraints are satisfied, the call-time actual-parameter arguments and global-variable values are substituted for the tokens in the function's template, producing the function call's output heap, which is then reconciled with the input heap to the call. (This last step is discussed in the next section.)

Figure 8 gives an example of a template resulting from an analyzed function. The value of P_1 shows how free variables d and g reside in the local and global namespaces, respectively. The conditional's test generates the constraint that t_d *is* $\{\}$ — d's value is a dictionary — in the then-arm. Since d is dereferenced within the then-arm, the constraint is expanded into a conditional constraint, t_d *is* $\{\} \Rightarrow (t_d$ *is* $\{\text{"}b\text{"} : t_{d'}\})$ — if d's initial value is a dictionary, then it must possess a "b" field. The updates to $d[\text{"}c\text{"}]$ and $d[\text{"}b\text{"}]$ are expressed *in the heap, on fresh tokens*, as seen in P_5.

P_6's value shows how g is updated in the else arm; note the constraint that d is "not a dictionary" within the else arm. The negation can be expressed as a disjunction of all the type structures that are not dictionary structures (here, "string or closure") or can be left as is; see Section 8.

Finally, the function's output is defined by P_7. The output heap collects updates to both local and free variables. Precision is lost by joining the heaps at P_5 and P_6. (In Section 9 we see how to regain precision by annotating d with a user-defined type that lets us generate multiple templates for a function definition.) The output heap is used at function-call time to update the call-time

Function and flow equations:

```
def f(d):
    global g
    if isinstance(d,dict):
        d['c'] = g
        g = d['b']
        d['b'] = {}
    else:
        g = d
```

$$
\begin{aligned}
P_0 &= assign\ d\ t_d\ initialHeap_{n_1,n_0}\\
P_1 &= assignglobal\ g\ t_g\ P_0\\
P_{2t} &= makeTrue\ (isinstance\ d\ dict)\ P_1\\
P_3 &= assign\ (index\ d\ \text{"}c\text{"})\ g\ P_{2t}\\
P_4 &= assign\ g\ (index\ d\ \text{"}b\text{"})\ P_4\\
P_5 &= assign\ (index\ d\ \text{"}b\text{"})\ \{\}\ P_4\\
P_{2f} &= makeFalse\ (isinstance\ d\ dict)\ P_1\\
P_6 &= assign\ g\ d\ P_{2f}\\
P_7 &= P_5 \sqcup P_6
\end{aligned}
$$

The environment is $[n_1 :: n_0]$, where n_0 is the address of the global namespace. The initial heap is $[n_1 \mapsto \{\}!, n_0 \mapsto \{\}] \times [\cdots] \times ()$, where the second component maps defined functions to their templates and the third component collects constraints on the function's free variables. The worklist solution is

$$
\begin{aligned}
P_0 &= [n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{\}],\ [\cdots],\ ()\\
P_1 &= [n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{g : t_g\}],\ [\cdots],\ ()\\
P_{2t} &= [n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{g : t_g\}, t_d \mapsto \{\}],\ [\cdots],\ t_d\ is\ \{\} \Rightarrow ()\\
P_3 &= [n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{g : t_g\}, t_d \mapsto \{\text{"}c\text{"} : t_g\}],\ [\cdots],\\
&\qquad t_d\ is\ \{\} \Rightarrow ()\\
P_4 &= [n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{g : t_{d'}\}, t_d \mapsto \{\text{"}c\text{"} : t_g, \text{"}b\text{"} : t_{d'}\}],\ [\cdots],\\
&\qquad t_d\ is\ \{\} \Rightarrow (t_d\ is\ \{\text{"}b\text{"} : t_{d'}\})\\
P_5 &= [n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{g : t_{d'}\}, t_d \mapsto \{\text{"}b\text{"} : \ell'_0, \text{"}c\text{"} : t_g\}, \ell'_0 \mapsto \{\}!],\ [\cdots],\\
&\qquad t_d\ is\ \{\} \Rightarrow (t_d\ is\ \{\text{"}b\text{"} : t_{d'}\})\\
P_{2f} &= [n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{g : t_g\}],\ [\cdots],\ t_d\ is\ \sim\{\} \Rightarrow ()\\
P_6 &= [n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{g : t_d\},\ t_d\ is\ \sim\{\} \Rightarrow ()\\
P_7 &= [n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{g : \{t_{d'}, t_d\}\}, t_d \mapsto \{\text{"}b\text{"} : \ell'_0, \text{"}c\text{"} : t_g\}, \ell'_0 \mapsto \{\}!],\ [\cdots],\\
&\qquad t_d\ is\ \{\} \Rightarrow (t_d\ is\ \{\text{"}b\text{"} : t_{d'}\})
\end{aligned}
$$

The value of P_7, heap and constraints, constitute the function's template.

Fig. 8. Analysis of a function

heap, and the collected constraints are validated on the call-time arguments and call-time heap. We consider this next.

6 Function-call processing

A function's template is applied when the function is called. These steps are undertaken:

1. At the point of call, the arguments and global variables are checked against the constraints listed in the function's template. If a constraint cannot be validated, a mismatch error (type error) is generated. If any of the arguments or globals themselves contain tokens, the constraints are copied into the caller's constraint set.

2. The value of the function template's output heap is used to update the call-time heap: For each of the namespaces listed in the caller's environment, the values of the function's free variables overwrite the values in the call-time heap. Here, locally constructed objects can "escape" from the called function's body and can be tracked by the analysis (cf. "escape analysis [13]).

Here are two calls of the function that was analyzed in Figure 8. First, we have the call,

```
g = 'mm'
h = {'b':'n'}
f(h)
```

At the point of the call, `f(h)`, the call-time heap is

$$[\{n_0 \mapsto \{g : "mm", h : \ell_0\}!, \ell_0 \mapsto \{"b" : "n"\}!].$$

The call generates these two bindings to the function's tokens: $t_d = \ell_0$ and $t_g = "mm"$. With the bindings, the analysis validates that the input constraint, $t_d \; is \; \{\} \Rightarrow (t_d \; is \; \{"b" : t_{d'}\})$, holds true; in the process, the binding, $t_{d'} = "n"$, is established. The three bindings are applied to the function's output heap — recall that it is:

$$[n_1 \mapsto \{d : t_d\}!, n_0 \mapsto \{g : \{t_{d'}, t_d\}\}, t_d \mapsto \{"b" : \ell_0', "c" : t_g\}, \ell_0' \mapsto \{\}!]$$

the above heap is reconciled with the call-time heap, generating this heap that results from the completed function call:

$$[n_0 \mapsto \{g : \{"n", \ell_0\}, h : \ell_0, \}!, \ell_0 \mapsto \{"b" : \ell_0', \; "c" : "mm"\}, \ell_0' \mapsto \{\}!$$

The example shows the loss of precision regarding `g`'s value. Precision can be improved if the function template would generate an output heap for each execution path (or, each generated constraint).

Here is a second, example call:

```
g = 'mm'
f(g)
```

At the point of call, the call-time heap is $[\{n_0 \mapsto \{g : "mm"\}]$. The bindings, $t_g = "mm"$ and $t_d = "mm"$, validate the input constraint. The function's output heap, listed above, is reconciled with the input heap, yielding $[n_0 \mapsto \{g : \{\bot, "nn"\}\}]$, which equals

$$[n_0 \mapsto \{g : "nn"\}]$$

In this example, there is no binding to $t_{d'}$, since the token is a subpart of *dictionary* t_d, which does not appear in the output heap that results from this call. ℓ_0' does not appear for the same reason.

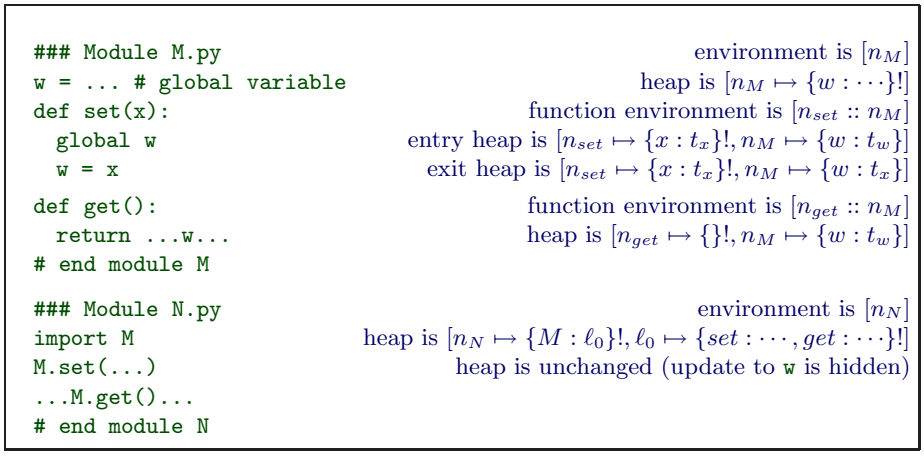

Fig. 9. Module definition, importation, and invocation

7 Module importation

Once a module is completely analyzed, a summary template must be generated from it. This is necessary because the caller of the module's functions owns an environment whose namespaces are disjoint from the module's. In particular, updates that a called function makes to a module's variables are hidden from the caller. Figure 9 shows an example module, `M.py`, and its importation and use by another, `N.py`. The environment used by module `N` is disjoint from module `M`'s. This means the call, `M.set(...)`, does not record the update to `set`, since `set` has already been flow-analyzed.

But this demands that `set`'s analysis must ensure that all possible arguments for parameter `x` meet the the input constraints generated by `set`'s body. (And global variable `w` must hold a value that always satisfies the input constraints.)

For these reasons, the analysis first constructs module `M`'s summary template by unioning and solving all constraints that were generated by all `M`'s functions on all `M`'s variables. The solution to the constraints defines each module variable's *invariant type*. Next, each function's output heap is checked to validate that the values of the global variables in the output heap satisfy the invariant types. (If a function fails to satisfy a variable's invariant type, that function cannot be called externally.)

The modular analysis of `M.py` generates a namespace of function templates that is imported by module `N` and used just like those in a local function call.

8 Filter functions

Within a scripting language, the tests in conditional commands play the role of classifying run-time values into types, and a useful analysis must exploit the information. To do so, the analysis interprets type-discriminating tests as

Define

$$makeTrueIsinstance : Denotable \times Heap \rightarrow Denotable \times Heap$$
$$makeTrueIsinstance(s \in String, \sigma) = \bot, \sigma$$
$$makeTrueIsinstance(\ell \in NSlocation, \sigma) = \ell, \sigma$$
$$makeTrueIsinstance(c \in CLlocation, \sigma) = \bot, \sigma$$
$$makeTrueIsinstance(t \in Token, \sigma) = t', \sigma + [t' \mapsto \{\}], \text{ where } t' \text{ is fresh}$$

For `instanceof(`L`, dict)`, say that θ is an L-value computed for L and that σ_0 is the current heap. The heap that results from the filter is defined

$$\begin{array}{l} let\ \{v_1, v_2, \cdots v_k\}_{k \geq 0} \\ let\ v'_{i+1}, \sigma_{i+1} = makeTrueIsinstance(v_{i+1}, \sigma_i), \text{ for all } 0 < i \leq k \\ let\ T = \{v"_i\}_{0 < i \leq k} \\ in\ update(\theta, T, \sigma_k) \end{array}$$

Fig. 10. Module definition, importation, and invocation

filter functions, which narrow (make more precise) the values that flow into the bodies of conditionals and loops. Figure 8 showed how the test predicate, `isinstance(d, dict)`, was used to update the value of the heap that entered the conditional's true arm to show that `d`'s value must be a dictionary (denoted, $\{\}$).

Figure 10 gives the definition used by the analysis for the `isinstance` predicate. For `instanceof(`L`, dict)`, the definition examines each of the denotable values that L denotes in the abstract heap. Each denotable value is filtered through `instanceof`; some proceed, some do not. In the case of a token, the token is specialized to a dictionary value. The results are collected into a set, which updates L's value.

The complement operation, $makeFalseIsinstance$, is defined similarly. In Figure 8, a "meta pattern", $\sim\{\}$, was used as the output for $makeFalseIsinstance$. This metapattern can be instantiated to a set of denotables, as suggested by Figure 10, or can be carried along as is. In the case of test predicates too complex to filter (e.g., $E_1 == E_2$), the filter function can default to the identity filter, which overrapproximates the flow into the arms of the conditional.

9 User-defined type equations and abstract parsing

User-defined types can improve the quality of an analysis; a user may write BNF-like type equations whose components are abstract-string and abstract-namespace values, e.g.,

```
N = {'s':N}! | 'nil'
```

A type equation defines a named denotable value that can be calculated upon by the flow analysis. Figure 11 displays a example function whose analysis utilizes the above named type, N. The named type generates a finite set of denotables,

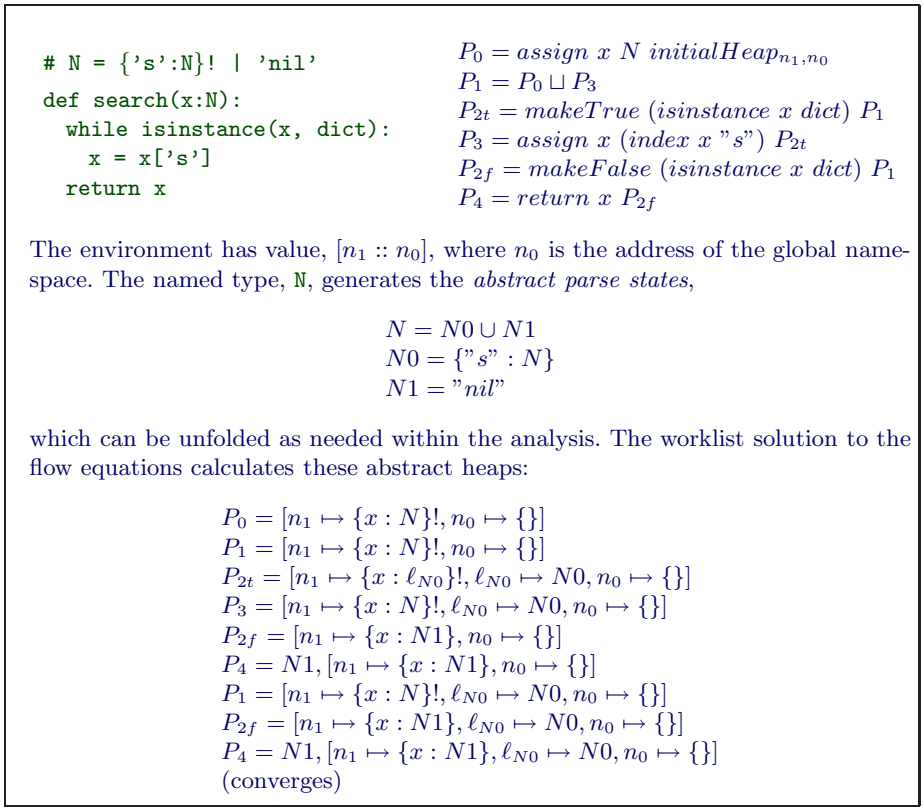

Fig. 11. Analysis of function with named parameter type

each of which is named by the parse state that is obtained by parsing a value with respect to the BNF equation. The parse states are used as denotable values by the analysis and are unfolded as needed (e.g., at the test, `isinstance(x,dict)`). In the example, parse state $N0$ acts as a "summary node," limiting the size of the abstract heap.

The type definitions also allow the analysis to generate multiple templates per function. For example, Figure 11 could be revised to generate an analysis of `def search(x:N0)` and an analysis of `def search(x:N1)`. The separate analyses prove useful in practice when a function is written to act as a disciminator, to be used in the test of a conditional command [3].

10 Implementation status

We have implemented a prototype analyzer that incorporates much (but not all) of the developments in this paper. The analyzer has proved to be a useful development and refinement tool for the techniques outlined here. The work is

ongoing, and we expect refinements of the technques described in this paper as a result of our profiling experiements.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.
2. R. Cartwright and M. Fagan. Soft typing. In *Proc. Conf. on Prog. Lang. Design and Implementation*, pages 278–292. ACM Press, 1991.
3. M. Felleisen. From Soft Scheme to Typed Scheme: Experiences from 20 years of script evolution, and some ideas on what works. Technical Report `www.ccs.neu.edu/home/matthias/Presentations/STOP/stop.pdf`, Keynote address, Workshop on Scripts to Programs, Genova, Italy, 2009.
4. M. Furr, J.-H. An, J. Foster, and M. Hicks. Static type inference for Ruby. In *Proc. Symposium on Applied Computing*, pages 1859–1866. ACM Press, 2009.
5. S.H. Jensen, A. Møller, and P. Thiemann. Type inference for Javascript. In *Proc. Static Analysis Symposium*, pages 238–255. Springer LNCS 5673, 2009.
6. D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. Symp. on Principles of Prog. Lang.*, pages 77–88. ACM Press, 1989.
7. S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Programming Lang. and Systems*, 24:217–298, 2002.
8. J. Siek and W. Taha. Gradual typing for functional languages. In *Proc. Scheme and Functional Programming Workshop*, pages 374–388. ACM Press, 2006.
9. J. Siek and W. Taha. Gradual typing for objects. In *Proc. European Conf. on Object-Oriented Programming*, pages 2–27. Springer LNCS 4609, 2007.
10. S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *Proc. OOPSLA Companion*, pages 964–974. ACM Press, 2006.
11. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Proc. Symp. Principles of Prog. Languages*, pages 395–406. ACM Press, 2007.
12. M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. Symp. on Logic in Computer Science*, pages 92–97. IEEE Press, 1989.
13. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. OOPSLA'99*, pages 187–206. ACM Press, 2009.