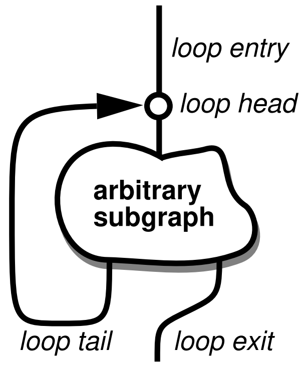
Chapter 11 Type Analysis and Splitting of Loops

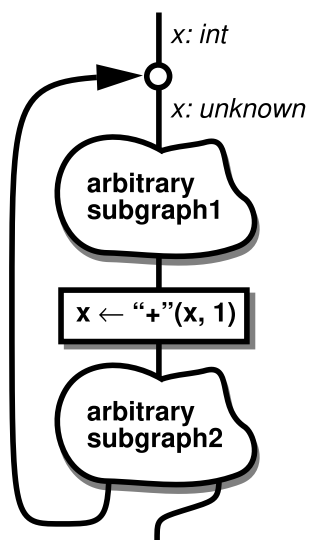
Loops pose special problems for type analysis. The basic problem is that the loop head is a kind of merge node, but the type information for some of the predecessors of the loop head (namely, the loop tail backwards branch) depends on the type information of the loop head, creating a circularity in the type analysis.



This chapter describes type analysis in the presence of loops and discusses synergistic interactions between loop type analysis and splitting.

11.1 Pessimistic Type Analysis

One approach to breaking the circularity would assume the most general possible value (the unknown value) and type (the unknown type) at the head of the loop for all names potentially assigned within the loop. These bindings would be guaranteed to be compatible with whatever bindings are subsequently computed for the loop tail, and so the type analysis would remain conservative and always produce legal control flow graphs. For example, in the following graph, the type of **x** would be automatically generalized to the unknown type at the loop head since there is an assignment to **x** within the loop:



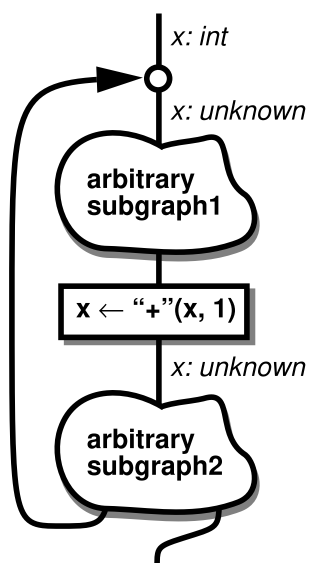
Unfortunately, this approach, which we call *pessimistic type analysis*, would sacrifice type information at precisely those places in the program that need the information the most to generate the best possible code: the inner loops of programs.

11.2 Traditional Iterative Data Flow Analysis

Traditional compilers resolve similar circularities in data flow analysis algorithms by performing the analysis *iteratively*. Iterative data flow analysis begins by assuming some data flow information for the loop head (usually just the information computed for the loop entry branch), analyzes the loop body, and then recomputes the data flow information at the loop head, this time including the results for the loop tail. If the information is unchanged (the *fixed point* in the analysis has been reached), then the analysis is done. If the information for the loop head has changed, then the previous analysis results are incorrect and the compiler must reanalyze the loop body with the new loop head information. This iteration will eventually terminate with a final least fixed point if the domain of the computed information is a finite lattice (a partial ordering with unique least and greatest elements representing the best and worst possible fixed points) and if the data flow propagation functions are monotonically increasing.

As traditionally applied, iterative data flow analysis operates on a fixed control flow graph. The compiler constructs a control flow graph, performs iterative flow analysis to compute some property of interest for all nodes in the graph, and then performs some optimization or transformation of the graph based on the computed information. If other optimizations need to be applied to the modified control flow graph, then most data flow analysis frameworks require recomputing all interesting information from scratch based on the new control flow graph. Incrementally updating data flow information after some modification to the control flow graph is an open research problem.

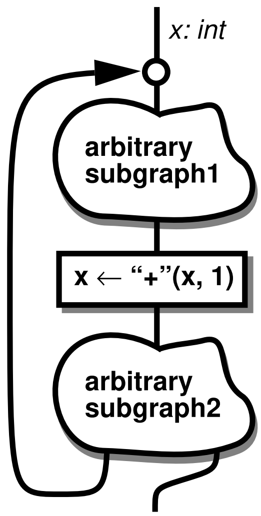
If the SELF compiler computed type information within loops by naively applying an iterative flow analyzer to the control flow graph prior to using the type information to perform optimizations such as inlining and splitting, then the computed type information would be very poor, virtually as bad as that computed using pessimistic type analysis. This surprising result is a consequence of SELF's use of messages for all computation, even computation as simple as arithmetic and instance variable accesses. If no messages within the loop are inlined, and an iterative flow analyzer is applied to compute the types of variables accessed within the loop, then any variables assigned the results of messages (including messages like **+**) must conservatively be assumed to be of unknown type, since the compiler would not know the type of the result of the non-inlined message. For example, naive iterative flow analysis would also compute that the type of **x** in the following graph must be unknown, since the type of the result of the non-inlined **+** message would be unknown:



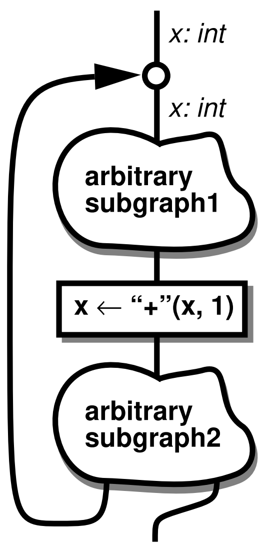
11.3 Iterative Type Analysis

The SELF compiler uses a technique we call *iterative type analysis* to compute precise types for variables modified in loops. Iterative type analysis is an extension of traditional iterative data flow analysis designed to cope with changing control flow graphs. As in iterative data flow analysis, the SELF compiler begins compiling a loop by assuming a certain set of types at the loop head. Unlike traditional flow analysis, however, the SELF compiler goes ahead and compiles the loop based on the initial assumed types. The SELF compiler's analysis may change the body of the loop as part of the analysis, such as by inlining messages or splitting apart sections of the loop body. Once the loop tail is reached, the types computed by the analysis for the loop tail are compared to the types assumed at the loop head. If they are compatible, then the loop tail is connected up to the loop head, and the compiler is done. Otherwise, the analysis must iterate, compiling a new version of the loop for the more general types.

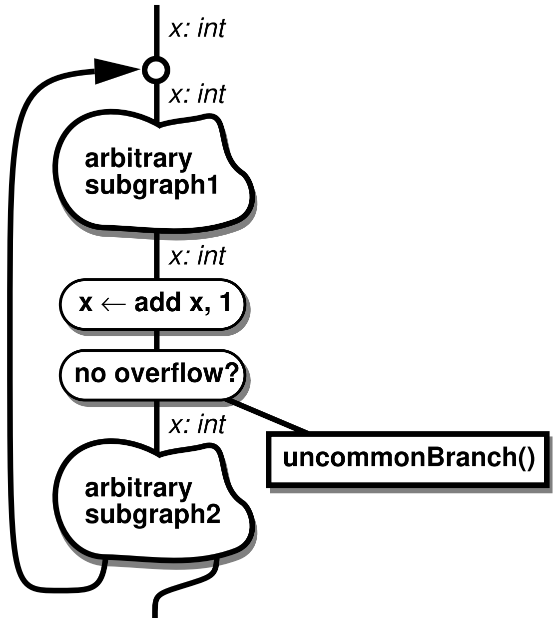
For example, when faced with the same control flow graph as before:



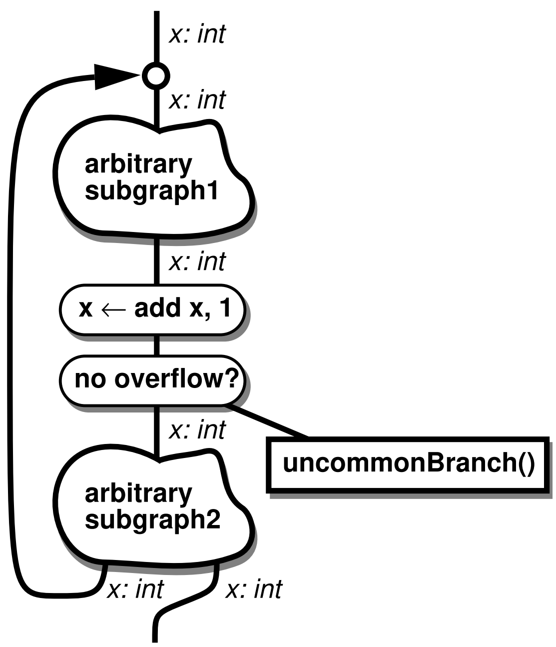
with iterative type analysis the compiler will first assume a set of types for the loop head derived from the loop entry types, in this case that **x** is an integer:



The compiler will then compile the body of the loop under these assumed types. When the compiler reaches the **+** message, its receiver, **x**, will be determined to be an integer (assuming no other assignments to **x** within the loop), and the compiler will inline the **+** message and the contained **intAdd** primitive to the following **add** machine instruction and overflow check:



After analyzing the rest of the control flow graph, **x** will be determined to be an integer at the loop tail, which is compatible with the types assumed at the loop head, and so the compiler is done:



The type information computed with this iterative type analysis is much better than the type information computed with either pessimistic type analysis or naive iterative data flow analysis. The **+** message has been inlined away with no extra overhead since iterative type analysis can compute that the type of **x** remains an integer throughout the loop. The other two approaches compute that the type of **x** is unknown at the top of the loop, and so at best a run-time type test would need to be inserted to check for an integer receiver of **+**. For other messages that are not be optimized using type prediction, the compiler would be prevented from inlining them at all without iterative type analysis.

The earlier description of the iterative type analysis algorithm purposely was vague on three points:

- What type information is assumed for the loop head initially?
- When is the type information computed for the loop tail compatible with the type information assumed at the loop head, allowing the loop tail to be connected with the loop head?
- How exactly does the analysis iterate if the loop tail type information is not compatible with the loop head type information?

Each of these questions has more than one reasonable answer, and different combinations of answers will result in different trade-offs among execution speed, compilation speed, and compiler simplicity. Iterative type analysis is thus revealed as a *framework* for a family of algorithms. The next three subsections provide the current SELF compiler's answers to these questions as well as some different answers from earlier SELF compilers. The question of compatibility will be taken up first, since the answer to this question impacts the answers for the other questions.

11.3.1 Compatibility

One central question is when the types computed at the loop tail are considered compatible with the types assumed at the loop head. One extreme approach would consider a loop tail type compatible with a loop head type if the loop head type *contains* the loop tail type (treating types as sets of values as described in section 9.1.4). This approach would make loop tails compatible with loop heads as often as possible, thus helping iterative analysis reach the fixed point quickly and saving compile time. Unfortunately, the approach also could sacrifice much of the type information available at the loop tail in the form of precise types by connecting a loop tail to a loop head compiled assuming much less specific type information. For example, if a variable were bound to the unknown type at the loop head and to the integer map type at a loop tail, then this compatibility rule would consider these two bindings compatible and allow the loop tail to be connected to the loop head. While this is legal, it would sacrifice type information available at the loop tail that could lead to better generated code if the loop tail were not connected to the loop head. We want a type compatibility rule that will not sacrifice this much of the compiler's hard-won type information.

The opposite extreme position on compatibility would only consider two types compatible if they are exactly the same. This position would avoid any loss of type information, since the loop tail must have exactly the same type information as a loop head to connect to it, but it could easily lead to many iterations as part of iterative type analysis. For example, there are $2^{2^{30}} - 1$ different types relating only to integers (the power set of the 2^{30} different integer constants, less the empty set), and type analysis could iterate this many times with such a definition of compatibility. Thus it is impractical to define type compatibility this narrowly.

The current SELF compiler uses a type compatibility rule midway between these two extreme positions. A type at a loop tail is considered compatible with a type at a loop head if the loop head type contains the loop tail type, and the loop head type does not lose any map-level type information. If the loop tail type is a constant type or an integer subrange type, then the loop head type can be no more general than the enclosing map type to be compatible. If the loop tail type is a union type whose components have different maps, then the loop head type can be no more general than the union of the corresponding map types. This compatibility rule implements the heuristic that map-level type information is the most important kind of type information and provides the lion's share of optimization opportunities, since map-level type information is the most general kind that still supports static binding and inlining of messages. Also, since in practice the compiler does not encounter more than a few different map types for a single variable,* the fixed point in the iterative type analysis is reached fairly quickly.

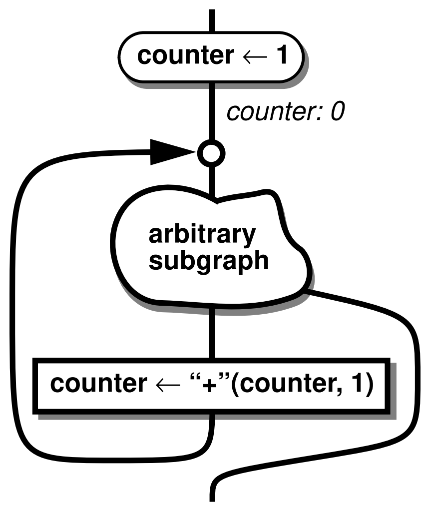
This strategy is amended in situations in which both common-case and uncommon-case branches exist, such as in uncommon branch extension methods (described in section 10.5) or when simulating a version of the system without lazy compilation of uncommon branches. Since the compiler is much more conservative when compiling uncommon branches, connecting a common-case loop tail to an uncommon-case loop head would also sacrifice the performance of that common-case path. Therefore, the compiler considers a loop tail compatible with a loop head only when the weight of the loop head is at least as great as the weight of the loop tail. Additionally, to minimize compilation time spent on uncommon branches, the compiler uses the most conservative strategy for type compatibility, considering an

* This is true mainly because currently there are only a few ways that the compiler can infer map-level type information. This characteristic may not be true in the future with the introduction of adaptive recompilation using polymorphic in-line caches [HCU91], and so this type compatibility rule may need to be revised to keep down the number of iterations needed to reach the fixed point.

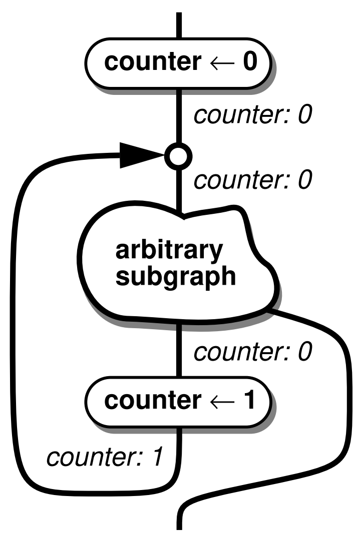
uncommon-case loop tail compatible with an uncommon-case loop head if the types at the loop head merely contain the types at the loop tail. Along uncommon branches, conserving compilation time and space is more important than preserving map-level type information.

11.3.2 Initial Loop Head Type Information

Another question to be answered is exactly what types are assumed initially at the loop head. An obvious strategy would assume that the loop head's type information was the same as the loop entry's type information. This approach is simple and precise: beginning with the loop entry's type information would enable the compiler to compute the most precise fixed points. Unfortunately, this approach frequently would cause the algorithm to iterate at least once, since in most cases at least one name would be assigned a type in the body of the loop different from that initially assumed at the entrance to the loop. For example, the following loop is typical of a standard **`for`** loop written in SELF, with a loop counter initialized to zero and incremented at the end of the body of the loop:

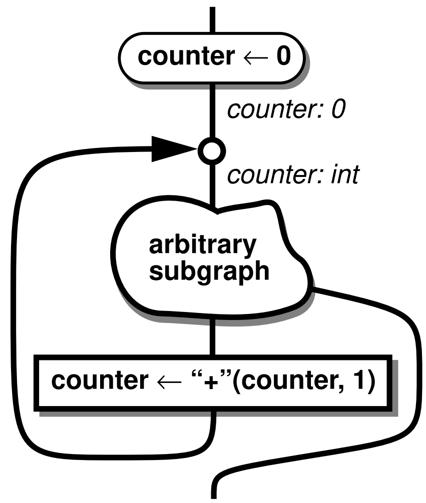


If the loop head types were assumed to be the same as the loop entry types, then after one iteration (and inlining the **`+`** message and constant-folding the **`intAdd`** primitive) the compiler would produce this graph:

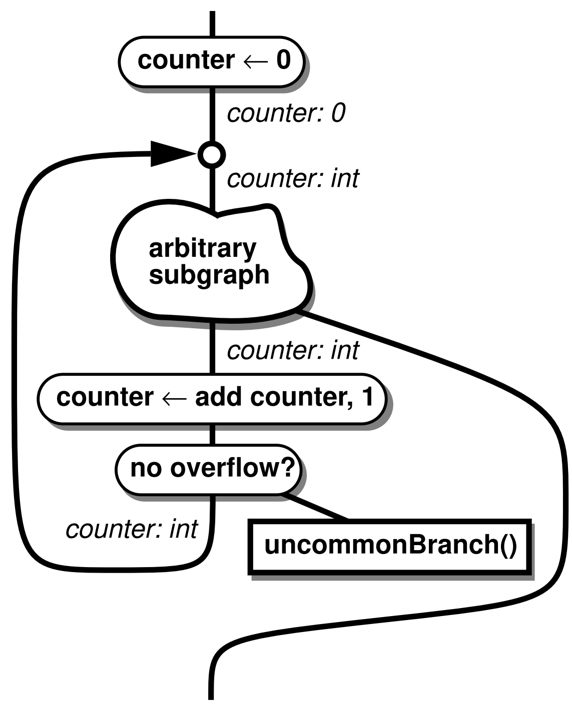


The type of **`counter`** computed at the loop tail would not be compatible with the type assumed at the loop head, since the type assumed at the loop head would not contain the corresponding type at the loop tail. Consequently, the analysis would be required to iterate with some more general type than the **`0`** constant type. Since most loops modify some loop counter from its initial constant value, this initial throw-away iteration would be expected for most loop structures.

Assignable local variables are likely to be assigned within the body of the loop a value different from the initial value computed at the entrance to the loop. To avoid an extra iteration, the current SELF compiler *generalizes* the types of assignable names at the loop head over the corresponding types at the loop entrance. To preserve map-level type information, this generalization is only to the enclosing map type (or union of map types if the loop entrance type is a union type). In the above **`for`** loop example, the compiler will generalize the initial type of the loop counter to the integer map type:



Then after compiling the loop body the compiler will reach the following graph:



The type at the loop tail is now compatible with the type at the loop head on the first iteration; no additional iterations are needed.

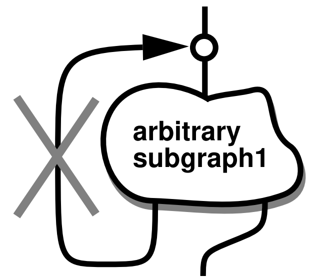
This generalization heuristic works well in the cases in which the assignments to local variables are of the same map as the initial value, such as is the case with integer loop counters. It also usually saves an iteration in iterative type analysis over the version of iterative type analysis in the earlier compiler. However, there are some situations in which this generalization of all assignable names is overly pessimistic. Some variables may be assignable but not actually assigned within the loop being analyzed; generalizing their types may lose type information. For example, in a doubly-nested loop, both loop counters are assignable, but the outer loop counter is not assigned within the inner loop. At the inner loop head the current SELF compiler will generalize the type of the outer loop counter from some integer subrange type (computed after the initial comparison against the outer loop's upper bound) to the more general integer map type. This unnecessary generalization can sacrifice possibilities for optimization, such as applying integer subrange analysis to eliminate unneeded array bounds checks or arithmetic overflow checks involving the outer loop counter. A better approach that would not lose this kind of type information would determine which variables might be assigned within a particular loop and only generalize those variables. This analysis is complicated by the fact that most of these assignments occur within blocks making up the body of the loop control structure, and accurately determining which variables might be assigned within some loop would require computing some sort of transitive closure of all blocks that could be invoked by messages sent within the loop. If implemented, this approach would solve the doubly-nested loop problem.

To minimize compilation time for uncommon-case loops, the SELF compiler generalizes the types of assignable variables all the way to the unknown type at uncommon-case loop heads, instead of just the enclosing map type as with common-case loop heads. In conjunction with the lenient rules for type compatibility of uncommon-case loop tails, this generalization ensures that an uncommon-case loop tail always connects to an uncommon-case loop head on the first pass; no iterations are needed for uncommon-case loops.

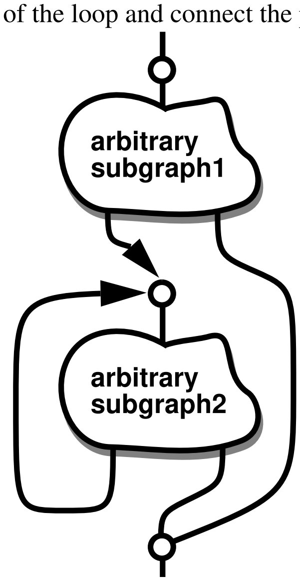
11.3.3 Iterating the Analysis

The final question remaining to be answered to complete the description of iterative type analysis is what happens when the loop tail is not compatible with a loop head, forcing the analysis to iterate. In traditional iterative data flow analysis, the information previously computed about the nodes in the loop is thrown away, new information is computed for the loop head by combining the information assumed at the previous iteration with the information computed at the loop tail, and finally the information for the body of the loop is recomputed based on the new, more general loop head information. This approach cannot be directly applied to iterative type analysis, since as part of analyzing the body of the loop the control flow graph is optimized based on the information. If the type information assumed at the loop head is determined to be overly optimistic, the optimizations performed based on that information are no longer valid. Backing out of optimizations like inlining and splitting would be very difficult.

The approach taken in the SELF compiler's iterative type analysis is to create a fresh new copy of the part of the control flow graph representing the loop body, and reapply type analysis to this fresh unoptimized copy. The fresh loop head is simply connected downstream of the previous iteration's incompatible loop tail, in effect *unrolling* the loop for the new type information. For example, if after reaching the loop tail on the first iteration it proves incompatible with the loop head:



the compiler will simply create a new copy of the loop and connect the previous copy's loop tail to it:



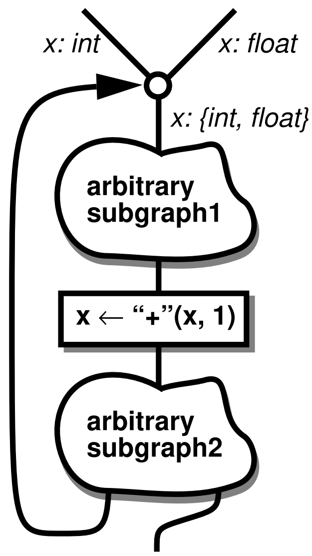
Unrolling loop bodies this way has important fringe benefits. Type tests and other code can get "hoisted" out of the normal loop body into earlier versions. For example, if the type of some variable is unknown at the top of the initial loop body, then the first loop head will assume the type of the variable is unknown. If the variable is treated like an integer inside the loop, such as by sending it **+** messages, then the compiler will use type prediction to insert a run-time type test and use splitting to compile a separate path through the loop body where the variable is known to be bound to an integer. When the loop tail is reached, the integer type will not be compatible with the unknown type at the loop head type, and so a separate version of the loop body will be compiled just for when the variable is bound to an integer. This second version of the loop will contain no type tests, since through the loop body the variable will be bound to an integer. If this turns out to be the common case (as is expected), then control will pass through the initial version of the loop once and thereafter remain in the faster integer-specific version of the loop.

This unrolling approach is quite simple to implement. However, this strategy may waste some compiled code space (and hence compile time) if the unrolled copies are very similar. Some compiled code space is reclaimed by generalizing the types of assignable names before the first iteration, as described in section 11.3.2, thus usually saving an iteration and correspondingly a whole copy of the loop body. An alternative approach would replace the original overly optimized loop body with the fresh new loop body, redirecting the predecessors of the original loop head to flow instead into the fresh new loop head. This would allow the compiler to conserve compiled code space and compile time by sharing similar parts of the loop body across what would have been separate copies. Unfortunately, this alternative approach would be more complex to implement and would require some mechanism to ensure that the new loop head starts out with more general type information that just what its predecessors indicate. Path-based type information exacerbates both of these problems. Future research could explore designs for iterating the type analysis that conserve compiled code space and compile time, work with path-based type information, and minimize compiler complexity.

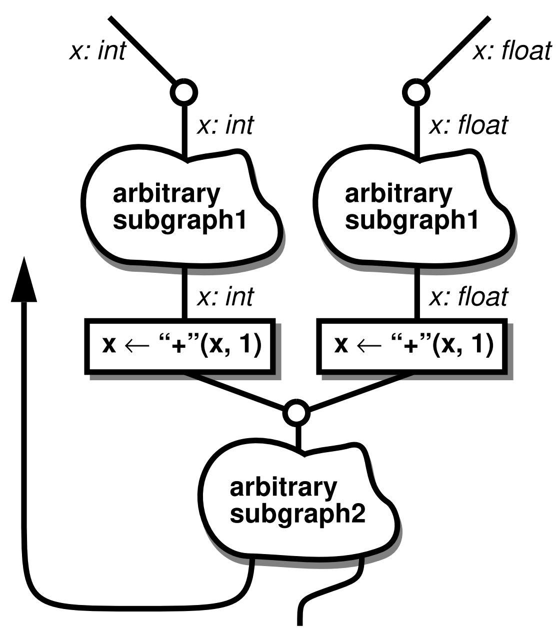
11.4 Iterative Type Analysis and Splitting

Iterative type analysis of loops and splitting have been carefully crafted to enable the SELF compiler to compile *multiple versions* of a loop, each version optimized for different combinations of run-time types. The inner loops of programs are the most important to optimize well, and by compiling separate versions of loops for different run-time types the SELF compiler is able to generate very good code for these loops.

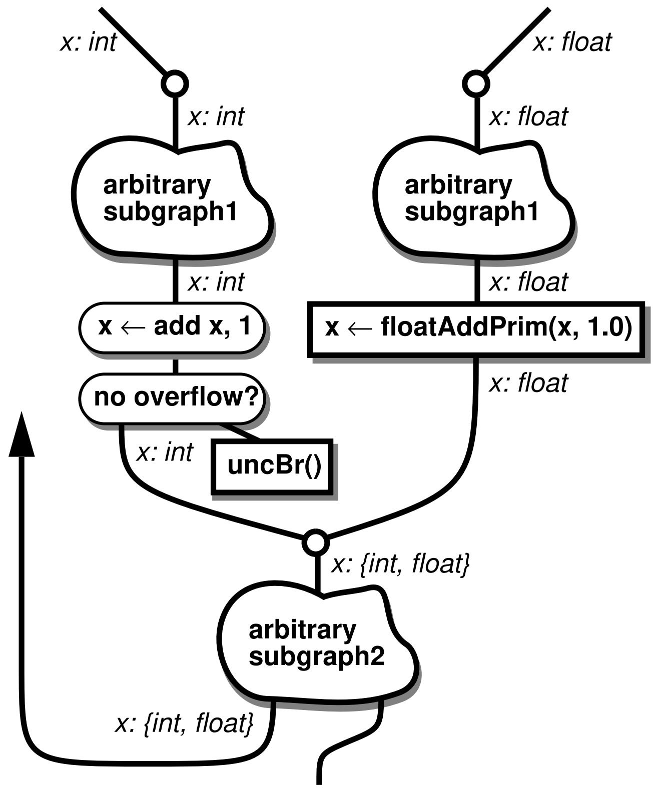
Since loop heads are just a special kind of merge node, they can be split apart just like merge nodes. For example, if two branches merge together at a loop head, and along one branch a variable has type integer, and along the other branch the variable has type floating point number:



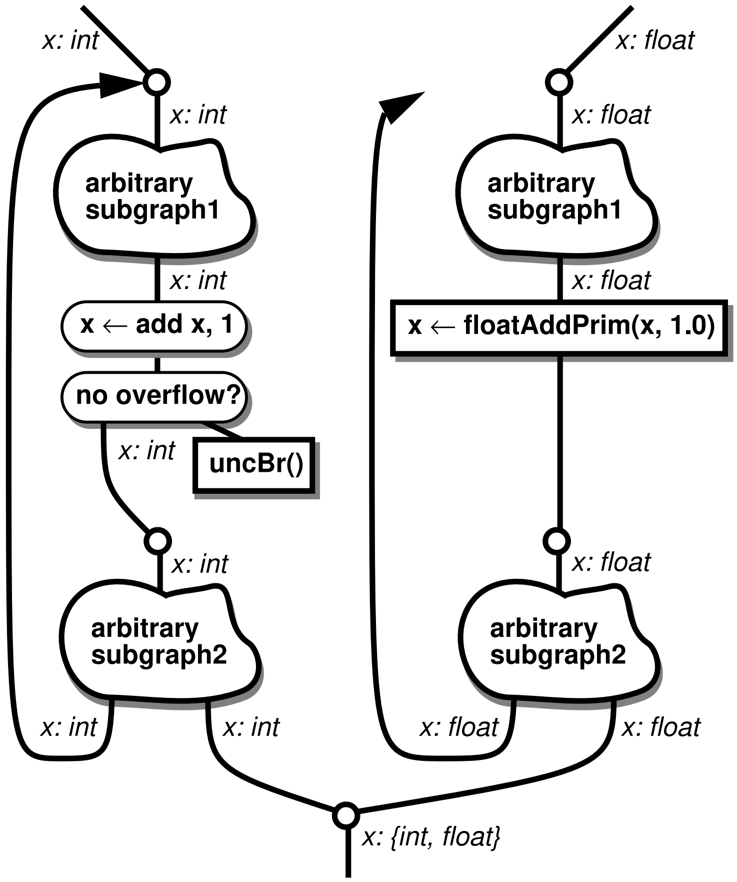
the compiler can split apart the loop head merge node if the variable is sent a message within the loop:



This splitting can create multiple loop heads for a single loop, each loop head for different types. The current SELF compiler's unrolling approach to iteration also creates multiple loop heads each with different types. Consequently, a loop tail may have several loop heads from which to choose when trying to connect to a type-compatible loop head. The compiler tries to find the highest-weight loop head with the most compatible types to connect a loop tail to. If no loop head is compatible, then the compiler tries to split the loop tail to match some loop head. For example, after inlining away each of the **+** messages and then compiling the rest of the loop the compiler reaches the following graph:

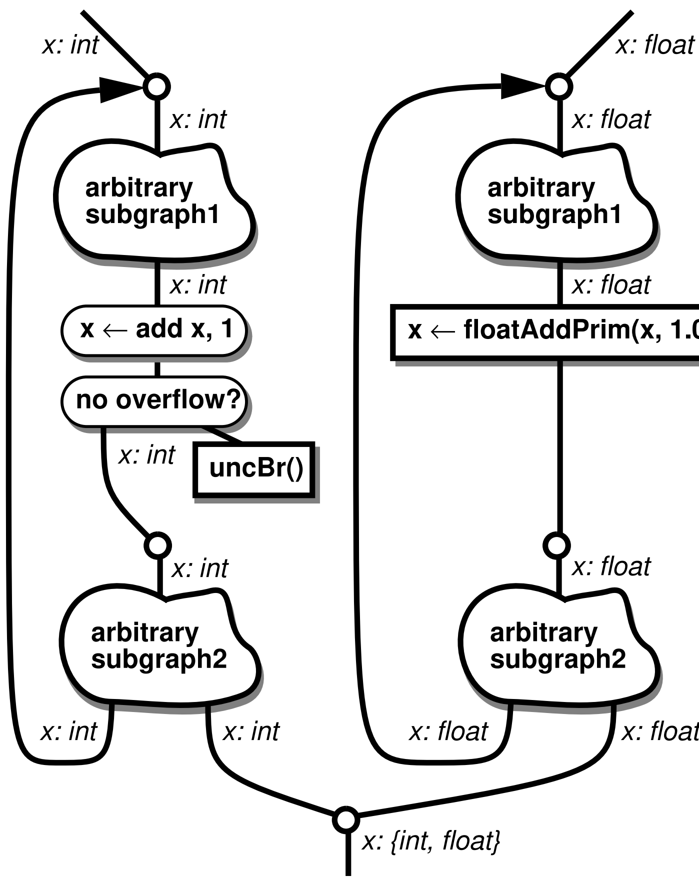


The loop tail type *{int, float}* does not directly match either of the loop head types. So the compiler tries to find a subset of the paths through the loop tail that can be split off and connected to a loop head. The compiler determines that the left-hand path producing the integer type binding would be compatible with the left-hand loop head if split off, so the compiler splits the loop tail accordingly (assuming that the cost of splitting is low enough) and connects the split loop tail to get the following graph:



After splitting a loop tail and connecting one of the copies, the compiler turns its attention to the remaining loop tail copy. This loop tail is itself checked against the available loop heads, and possibly split again if it cannot be connected directly to a loop head. This process continues until either the loop tail is successfully connected to a loop head, in which case the compiler has finished compiling the loop, or the loop tail cannot be either split or connected to a loop head, in which case the compiler falls back on its unrolling strategy, creating a fresh copy of the loop (and a new loop head) for the loop tail's combination of types. This process of splitting loop tails to match loop heads is guaranteed to terminate with either a successful connection or by unrolling the loop since at each split the number of paths in the leftover loop tail decreases.

In the case of the example loop, the compiler determines that the leftover loop tail is compatible with the right-hand loop head and connects it up to get this control flow graph:



This effect of splitting loop heads and subsequently splitting loop tails often leads the compiler to generate multiple versions of a single source loop, each compiled version for different type bindings. In the above example, the compiler has produced two completely independent compiled versions of the original source loop, with one version for the case where **x** is an integer and a separate version for the case where **x** is a floating point number. Each version of the loop is optimized for a particular combination of types, and so can be much faster than a single general version of the loop could be. In the above example, if the compiler were constrained to generate only a single version of the loop, then the compiler would be forced to use type casing to optimize the **+** message for the integer and floating point cases, introducing extra run-time overhead for the type test. This extra overhead can become significant as the number of operations in the loop that could be split apart increases.

This splitting of loops for different types enables the SELF compiler to compete with optimizing compilers for traditional languages in execution performance without sacrificing the extra power available to SELF programmers, such as pure message passing and generic arithmetic support. Typically, one version (or sometimes a few versions) of a loop will be split off and optimized for the common-case types; these types include those that would be used in the equivalent program in a traditional language, such as integers and fixed-length arrays. This common-case version of the loop can achieve quite good performance, since the compiler is compiling a version of the loop specific to those common-case types. The SELF compiler gets nearly the same information that is available to the compiler for the traditional language and hence can generate code that runs nearly as fast as that generated by the compiler for the traditional language. The SELF compiler will also generate an extra version of the loop to handle any other types or situations that might arise that do not fall into the category of common-case types or situations; this extra version of the loop implements the additional semantics available in SELF in the form of pure message passing and generic arithmetic. With lazy compilation, this extra version will be in the uncommon branch extension method and usually is never actually compiled at all. The separation of the common-case version(s) from the uncommon-case version(s) is the primary mechanism used by the SELF compiler to rival the performance of traditional languages.

11.5 Summary

The SELF compiler uses iterative type analysis to infer relatively precise types in the presence of loops. This algorithm performs optimizations such as inlining and splitting on the body of the loop as part of each iteration in the analysis. As a result, the compiler computes much more precise type information than would be possible with a standard data flow analysis algorithm. The SELF compiler uses various heuristics to reach the fixed point as quickly as possible without sacrificing a significant amount of type information. These heuristics include automatically generalizing the types of assignable names to the enclosing map type at loop heads and connecting loop tails to loop heads as long as the loop head does not sacrifice map-level type information. When a loop tail does not match any available loop head, the loop is unrolled by appending a fresh copy of the loop body to the loop tail and continuing the analysis.

The SELF compiler treats loop heads like merge nodes and will split a loop head apart if a node downstream of the loop head wants some type information that was diluted by either the loop head merge node or by a merge node before the loop head. This splitting creates multiple loop heads from which to choose when connecting a loop tail to a loop head. Unrolling a loop when a loop tail is not compatible with any available loop head also creates several loop heads from which a loop tail can choose. A loop tail itself can be split apart when it does not directly match any loop head, but a subset of the paths that reach the loop tail do match a loop head. With both loop heads and loop tails being split apart, the SELF compiler frequently ends up creating several completely independent versions of a loop, with each loop head leading to a matching loop tail. Each of these versions can be much faster than could a single general version of the loop, and consequently generating multiple versions of loops each for different run-time types is one of the key sources of SELF's execution speed. When combined with lazy compilation of uncommon branches, the SELF compiler frequently compiles one or two common-case versions of a loop to handle the standard situations, with an additional general version of the loop compiled only if and when needed.